REMARKS

I. Introduction

Claims 1-48 are pending in the above application.

Claims 1-17 stand rejected under 35 U.S.C. § 101 under the judicially created doctrine of obvious-type double patenting.

Claims 1, 7, 13, 15 and 16 stand rejected under 35 U.S.C. § 103.

Claims 2-6, 8-12, 14 and 17 have not been rejected under prior art and are presumed to be allowable, but for the obvious double patenting rejection thereon.

Claims 18-48 are newly added.

Claims 1, 7, 13, 15 and 18 are independent.

II. Amendments

A. Substitute Specification

The specification has been amended to clarify that priority under 35 U.S.C. § 120 is being claimed to U.S. Application No. 08/754,827, now U.S. Patent No. 5,822,603 in addition to U.S. Application Nos. 09/382,402, 09/169,963, and 08/516,036 and to the priority claim under 35 U.S.C. § 119(e) to Provisional Application 60/097,635. The priority claim has also been amended to provide the patent numbers for U.S. Application Nos. 09/382,402, 09/169,963, and 08/516,036.

The specification has also been amended to bodily incorporate subject matter from the Appendix attached to and forming part of the specification. Although Applicant makes no admission that the bodily incorporation of this subject is required under the USPTO procedures, Applicant has undertaken this task in order to assist the Examiner in the examination of new

B

claims 18-48. See, MPEP 608.01(p). A declaration is also provided from the Applicant stating that the amendatory material consists of the same material contained in the Appendix. Applicants note that a paper copy of the Appendix is also provided with this response for the Examiner's review.

No new matter has been added to the specification or the abstract.

B. Drawings

Figures 12A-34E have been added. Each of Figures 12A-34E have been bodily incorporated from the Appendix attached to and forming part of the specification. Although Applicant makes no admission that the bodily incorporation of this subject is required under the USPTO procedures, Applicant has undertaken this task in order to assist the Examiner in the examination the newly added claims. See, MPEP 608.01(p). A declaration is also provided from the Applicant stating that the new Figures 12A-34E (the amendatory material) consists of the same material contained in the Appendix.

No new matter has been added to the drawings.

C. Claims 18-48

New claims 18-48 are believed to recite features not disclosed or suggested by the prior art. Accordingly, new claims 18-48 are believed to be in condition for allowance.

No new matter has been added in new claims 18-48.

B

Definition

```
def Group(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.ADD:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← c_{i+size-1..i} + b_{i+size-1..i}
            endfor
        G.ADD.L:
            for i ← 0 to 128-size by size
                t ← (c_{i+size-1} || c_{i+size-1..i}) + (b_{i+size-1} || b_{i+size-1..i})
                a_{i+size-1..i} ← (t_size ≠ t_{size-1}) ? (t_size || t_{size-1}^{size-1}) : t_{size-1..0}
            endfor
        G.ADD.L.U:
            for i ← 0 to 128-size by size
                t ← (0^1 || c_{i+size-1..i}) + (0^1 || b_{i+size-1..i})
                a_{i+size-1..i} ← (t_size ≠ 0) ? (1^size) : t_{size-1..0}
            endfor
        G.ADD.O:
            for i ← 0 to 128-size by size
                t ← (c_{i+size-1} || c_{i+size-1..i}) + (b_{i+size-1} || b_{i+size-1..i})
                if t_size ≠ t_{size-1} then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← t_{size-1..0}
            endfor
        G.ADD.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || c_{i+size-1..i}) + (0^1 || b_{i+size-1..i})
                if t_size ≠ 0 then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← t_{size-1..0}
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Fig. 26C

Operation codes

G.SET.AND.E.8	Group set and equal zero bytes
G.SET.AND.E.16	Group set and equal zero doublets
G.SET.AND.E.32	Group set and equal zero quadlets
G.SET.AND.E.64	Group set and equal zero octlets
G.SET.AND.E.128	Group set and equal zero hexlet
G.SET.AND.NE.8	Group set and not equal zero bytes
G.SET.AND.NE.16	Group set and not equal zero doublets
G.SET.AND.NE.32	Group set and not equal zero quadlets
G.SET.AND.NE.64	Group set and not equal zero octlets
G.SET.AND.NE.128	Group set and not equal zero hexlet
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.E.128	Group set equal hexlet
G.SET.GE.8	Group set greater equal signed bytes
G.SET.GE.16	Group set greater equal signed doublets
G.SET.GE.32	Group set greater equal signed quadlets
G.SET.GE.64	Group set greater equal signed octlets
G.SET.GE.128	Group set greater equal signed hexlet
G.SET.GE.U.8	Group set greater equal unsigned bytes
G.SET.GE.U.16	Group set greater equal unsigned doublets
G.SET.GE.U.32	Group set greater equal unsigned quadlets
G.SET.GE.U.64	Group set greater equal unsigned octlets
G.SET.GE.U.128	Group set greater equal unsigned hexlet
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.L.128	Group set signed less hexlet
G.SET.L.U.8	Group set less unsigned bytes
G.SET.L.U.16	Group set less unsigned doublets
G.SET.L.U.32	Group set less unsigned quadlets
G.SET.L.U.64	Group set less unsigned octlets
G.SET.L.U.128	Group set less unsigned hexlet
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets
G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets
G.SET.NE.128	Group set not equal hexlet
G.SUB.8	Group subtract bytes
G.SUB.8.O	Group subtract signed bytes check overflow

Fig. 27A

G.SUB.16	Group subtract doublets
G.SUB.16.O	Group subtract signed doublets check overflow
G.SUB.32	Group subtract quadlets
G.SUB.32.O	Group subtract signed quadlets check overflow
G.SUB.64	Group subtract octlets
G.SUB.64.O	Group subtract signed octlets check overflow
G.SUB.128	Group subtract hexlet
G.SUB.128.O	Group subtract signed hexlet check overflow
G.SUB.L.8	Group subtract limit signed bytes
G.SUB.L.16	Group subtract limit signed doublets
G.SUB.L.32	Group subtract limit signed quadlets
G.SUB.L.64	Group subtract limit signed octlets
G.SUB.L.128	Group subtract limit signed hexlet
G.SUB.L.U.8	Group subtract limit unsigned bytes
G.SUB.L.U.16	Group subtract limit unsigned doublets
G.SUB.L.U.32	Group subtract limit unsigned quadlets
G.SUB.L.U.64	Group subtract limit unsigned octlets
G.SUB.L.U.128	Group subtract limit unsigned hexlet
G.SUB.U.8.O	Group subtract unsigned bytes check overflow
G.SUB.U.16.O	Group subtract unsigned doublets check overflow
G.SUB.U.32.O	Group subtract unsigned quadlets check overflow
G.SUB.U.64.O	Group subtract unsigned octlets check overflow
G.SUB.U.128.O	Group subtract unsigned hexlet check overflow

Fig. 27A (cont'd)

Format

G.op.size rd=rb,rc

rd=gopsize(rb,rc)

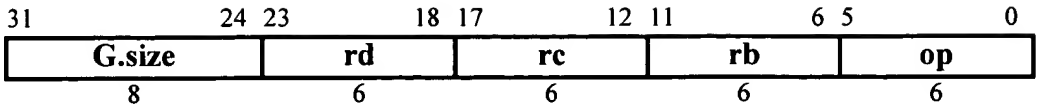


Fig. 27B

Definition

```
def GroupReversed(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.SUB:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← b_{i+size-1..i} - c_{i+size-1..i}
            endfor
        G.SUB.L:
            for i ← 0 to 128-size by size
                t ← (b_{i+size-1} || b_{i+size-1..i}) - (c_{i+size-1} || c_{i+size-1..i})
                a_{i+size-1..i} ← (t_size ≠ t_{size-1}) ? (t_size || t_{size-1}^{size-1}) : t_{size-1..0}
            endfor
        G.SUB.LU:
            for i ← 0 to 128-size by size
                t ← (0^1 || b_{i+size-1..i}) - (0^1 || c_{i+size-1..i})
                a_{i+size-1..i} ← (t_size ≠ 0) ? 0^size : t_{size-1..0}
            endfor
        G.SUB.O:
            for i ← 0 to 128-size by size
                t ← (b_{i+size-1} || b_{i+size-1..i}) - (c_{i+size-1} || c_{i+size-1..i})
                if (t_size ≠ t_{size-1}) then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← t_{size-1..0}
            endfor
        G.SUB.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || b_{i+size-1..i}) - (0^1 || c_{i+size-1..i})
                if (t_size ≠ 0) then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← t_{size-1..0}
            endfor
        G.SET.E:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← (b_{i+size-1..i} = c_{i+size-1..i})^size
            endfor
        G.SET.NE:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← (b_{i+size-1..i} ≠ c_{i+size-1..i})^size
            endfor
        G.SET.AND.E:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((b_{i+size-1..i} and c_{i+size-1..i}) = 0)^size
            endfor
```

Fig. 27C

```
            G.SET.AND.NE:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← ((b_{i+size-1..i} and c_{i+size-1..i}) ≠ 0)^size
                endfor
            G.SET.L:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← ((rc = rb) ? (b_{i+size-1..i} < 0) : (b_{i+size-1..i} < c_{i+size-1..i}))^size
                endfor
            G.SET.GE:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← ((rc = rb) ? (b_{i+size-1..i} ≥ 0) : (b_{i+size-1..i} ≥ c_{i+size-1..i}))^size
                endfor
            G.SET.L.U:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← ((rc = rb) ? (b_{i+size-1..i} > 0) :
                        ((0 || b_{i+size-1..i}) < (0 || c_{i+size-1..i})))^size
                endfor
            G.SET.GE.U:
                for i ← 0 to 128-size by size
                    a_{i+size-1..i} ← ((rc = rb) ? (b_{i+size-1..i} ≤ 0) :
                        ((0 || b_{i+size-1..i}) ≥ (0 || c_{i+size-1..i})))^size
                endfor
        endcase
        RegWrite(rd, 128, a)
enddef
```

Fig. 27C (cont'd)

Operation codes

E.CON.8	Ensemble convolve signed bytes
E.CON.16	Ensemble convolve signed doublets
E.CON.32	Ensemble convolve signed quadlets
E.CON.64	Ensemble convolve signed octlets
E.CON.C.8	Ensemble convolve complex bytes
E.CON.C.16	Ensemble convolve complex doublets
E.CON.C.32	Ensemble convolve complex quadlets
E.CON.M.8	Ensemble convolve mixed-signed bytes
E.CON.M.16	Ensemble convolve mixed-signed doublets
E.CON.M.32	Ensemble convolve mixed-signed quadlets
E.CON.M.64	Ensemble convolve mixed-signed octlets
E.CON.U.8	Ensemble convolve unsigned bytes
E.CON.U.16	Ensemble convolve unsigned doublets
E.CON.U.32	Ensemble convolve unsigned quadlets
E.CON.U.64	Ensemble convolve unsigned octlets
E.DIV.64	Ensemble divide signed octlets
E.DIV.U.64	Ensemble divide unsigned octlets
E.MUL.8	Ensemble multiply signed bytes
E.MUL.16	Ensemble multiply signed doublets
E.MUL.32	Ensemble multiply signed quadlets
E.MUL.64	Ensemble multiply signed octlets
E.MUL.SUM.8	Ensemble multiply sum signed bytes
E.MUL.SUM.16	Ensemble multiply sum signed doublets
E.MUL.SUM.32	Ensemble multiply sum signed quadlets
E.MUL.SUM.64	Ensemble multiply sum signed octlets
E.MUL.C.8	Ensemble complex multiply bytes
E.MUL.C.16	Ensemble complex multiply doublets
E.MUL.C.32	Ensemble complex multiply quadlets
E.MUL.M.8	Ensemble multiply mixed-signed bytes
E.MUL.M.16	Ensemble multiply mixed-signed doublets
E.MUL.M.32	Ensemble multiply mixed-signed quadlets
E.MUL.M.64	Ensemble multiply mixed-signed octlets
E.MUL.P.8	Ensemble multiply polynomial bytes
E.MUL.P.16	Ensemble multiply polynomial doublets
E.MUL.P.32	Ensemble multiply polynomial quadlets
E.MUL.P.64	Ensemble multiply polynomial octlets
E.MUL.SUM.C.8	Ensemble multiply sum complex bytes
E.MUL.SUM.C.16	Ensemble multiply sum complex doublets
E.MUL.SUM.C.32	Ensemble multiply sum complex quadlets
E.MUL.SUM.M.8	Ensemble multiply sum mixed-signed bytes
E.MUL.SUM.M.16	Ensemble multiply sum mixed-signed doublets
E.MUL.SUM.M.32	Ensemble multiply sum mixed-signed quadlets
E.MUL.SUM.M.64	Ensemble multiply sum mixed-signed octlets

Fig. 28A

E.MUL.SUM.U.8	Ensemble multiply sum unsigned bytes
E.MUL.SUM.U.16	Ensemble multiply sum unsigned doublets
E.MUL.SUM.U.32	Ensemble multiply sum unsigned quadlets
E.MUL.SUM.U.64	Ensemble multiply sum unsigned octlets
E.MUL.U.8	Ensemble multiply unsigned bytes
E.MUL.U.16	Ensemble multiply unsigned doublets
E.MUL.U.32	Ensemble multiply unsigned quadlets
E.MUL.U.64	Ensemble multiply unsigned octlets

Fig. 28A (cont'd)

Format

E.op.size rd=rc,rb

rd=eopsize(rc,rb)

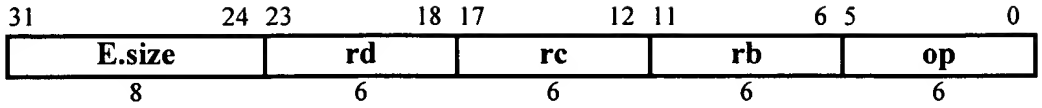


Fig. 28B

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
        mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def c ← PolyMultiply(size,a,b) as
        p[0] ← 0^{2*size}
        for k ← 0 to size-1
                p[k+1] ← p[k] ^ a_k ? (0^{size-k} || b || 0^k) : 0^{2*size}
        endfor
        c ← p[size]
enddef

def Ensemble(op,size,rd,rc,rb)
        c ← RegRead(rc, 128)
        b ← RegRead(rb, 128)
        case op of
                E.MUL:, E.MUL.C:, EMUL.SUM, E.MUL.SUM.C, E.CON, E.CON.C, E.DIV:
                        cs ← bs ← 1
                E.MUL.M:, EMUL.SUM.M, E.CON.M:
                        cs ← 0
                        bs ← 1
                E.MUL.U:, EMUL.SUM.U, E.CON.U, E.DIV.U, E.MUL.P:
                        cs ← bs ← 0
        endcase
        case op of
                E.MUL, E.MUL.U, E.MUL.M:
                        for i ← 0 to 64-size by size
                                d_{2*(i+size)-1..2*i} ← mul(size,2*size,cs,c,i,bs,b,i)
                        endfor
                E.MUL.P:
                        for i ← 0 to 64-size by size
                                d_{2*(i+size)-1..2*i} ← PolyMultiply(size,c_{size-1+i..i},b_{size-1+i..i})
                        endfor
                E.MUL.C:
                        for i ← 0 to 64-size by size
                                if (i and size) = 0 then
                                        p ← mul(size,2*size,1,c,i,1,b,i) - mul(size,2*size,1,c,i+size,1,b,i+size)
                                else
                                        p ← mul(size,2*size,1,c,i,1,b,i+size) + mul(size,2*size,1,c,i,1,b,i+size)
                                endif
                                d_{2*(i+size)-1..2*i} ← p
                        endfor
                E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M:
                        p[0] ← 0^{128}
                        for i ← 0 to 128-size by size
                                p[i+size] ← p[i] + mul(size,128,cs,c,i,bs,b,i)
                        endfor
                        a ← p[128]
                E.MUL.SUM.C:
                        p[0] ← 0^{64}
                        p[size] ← 0^{64}
                        for i ← 0 to 128-size by size
                                if (i and size) = 0 then
                                        p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i)
                                                          - mul(size,64,1,c,i+size,1,b,i+size)
                                else
                                        p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i+size)
                                                          + mul(size,64,1,c,i+size,1,b,i)
                                endif
                        endfor
                        a ← p[128+size] || p[128]
```

Fig. 28C

```
            E.CON, E.CON.U, E.CON.M:
                p[0] ← 0^128
                for j ← 0 to 64-size by size
                    for i ← 0 to 64-size by size
                        p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
                            mul(size,2*size,cs,c,i+64-j,bs,b,j)
                    endfor
                endfor
                a ← p[64]
            E.CON.C:
                p[0] ← 0^128
                for j ← 0 to 64-size by size
                    for i ← 0 to 64-size by size
                        if ((~i) and j and size) = 0 then
                            p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
                                mul(size,2*size,1,c,i+64-j,1,b,j)
                        else
                            p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i -
                                mul(size,2*size,1,c,i+64-j+2*size,1,b,j)
                        endif
                    endfor
                endfor
                a ← p[64]
            E.DIV:
                if (b = 0) or ( (c = (1||0^63)) and (b = 1^64) ) then
                    a ← undefined
                else
                    q ← c / b
                    r ← c - q*b
                    a ← r63..0 || q63..0
                endif
            E.DIV.U:
                if b = 0 then
                    a ← undefined
                else
                    q ← (0 || c) / (0 || b)
                    r ← c - (0 || q)*(0 || b)
                    a ← r63..0 || q63..0
                endif
        endcase
        RegWrite(rd, 128, a)
enddef
```

Fig. 28C (cont'd)

Floating-point function Definitions

```
def eb ← ebits(prec) as
    case pref of
        16:
            eb ← 5
        32:
            eb ← 8
        64:
            eb ← 11
        128:
            eb ← 15
    endcase
enddef

def eb ← ebias(prec) as
    eb ← 0 || 1^(ebits(prec)-1)
enddef

def fb ← fbits(prec) as
    fb ← prec – 1 – eb
enddef

def a ← F(prec, ai) as
    a.s ← ai_(prec-1)
    ae ← ai_(prec-2..fbits(prec))
    af ← ai_(fbits(prec)-1..0)
    if ae = 1^(ebits(prec)) then
        if af = 0 then
            a.t ← INFINITY
        elseif af_(fbits(prec)-1) then
            a.t ← SNaN
            a.e ← -fbits(prec)
            a.f ← 1 || af_(fbits(prec)-2..0)
        else
            a.t ← QNaN
            a.e ← -fbits(prec)
            a.f ← af
        endif
```

Fig. 29

```
        elseif ae = 0 then
                if af = 0 then
                        a.t ← ZERO
                else
                        a.t ← NORM
                        a.e ← 1-ebias(prec)-fbits(prec)
                        a.f ← 0 || af
                endif
        else
                a.t ← NORM
                a.e ← ae-ebias(prec)-fbits(prec)
                a.f ← 1 || af
        endif
enddef

def a ← DEFAULTQNAN as
        a.s ← 0
        a.t ← QNAN
        a.e ← -1
        a.f ← 1
enddef

def a ← DEFAULTSNAN as
        a.s ← 0
        a.t ← SNAN
        a.e ← -1
        a.f ← 1
enddef

def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
        if a.t=NORM and b.t=NORM then
                // d,e are a,b with exponent aligned and fraction adjusted
                if a.e > b.e then
                        d ← a
                        e.t ← b.t
                        e.s ← b.s
                        e.e ← a.e
                        e.f ← b.f || 0^(a.e-b.e)
                else if a.e < b.e then
                        d.t ← a.t
                        d.s ← a.s
                        d.e ← b.e
                        d.f ← a.f || 0^(b.e-a.e)
                        e ← b
                endif
                c.t ← d.t
                c.e ← d.e
                if d.s = e.s then
                        c.s ← d.s
                        c.f ← d.f + e.f
                elseif d.f > e.f then
                        c.s ← d.s
                        c.f ← d.f – e.f
```

Fig. 29 (cont'd)

```
            elseif d.f < e.f then
                c.s ← e.s
                c.f ← e.f – d.f
            else
                c.s ← r=F
                c.t ← ZERO
            endif
        // priority is given to b operand for NaN propagation
        elseif (b.t=SNAN) or (b.t=QNAN) then
            c ← b
        elseif (a.t=SNAN) or (a.t=QNAN) then
            c ← a
        elseif a.t=ZERO and b.t=ZERO then
            c.t ← ZERO
            c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
        // NULL values are like zero, but do not combine with ZERO to alter sign
        elseif a.t=ZERO or a.t=NULL then
            c ← b
        elseif b.t=ZERO or b.t=NULL then
            c ← a
        elseif a.t=INFINITY and b.t=INFINITY then
            if a.s ≠ b.s then
                c ← DEFAULTSNAN // Invalid
            else
                c ← a
            endif
        elseif a.t=INFINITY then
            c ← a
        elseif b.t=INFINITY then
            c ← b
        else
            assert FALSE // should have covered al the cases above
        endif
enddef

def b ← fneg(a) as
        b.s ← ~a.s
        b.t ← a.t
        b.e ← a.e
        b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

def c ← fcom(a,b) as
        if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
            c ← U
        elseif a.t=INFINITY and b.t=INFINITY then
            if a.s ≠ b.s then
                c ← (a.s=0) ? G: L
```

Fig. 29 (cont'd)

```
            else
                c ← E
            endif
        elseif a.t=INFINITY then
            c ← (a.s=0) ? G: L
        elseif b.t=INFINITY then
            c ← (b.s=0) ? G: L
        elseif a.t=NORM and b.t=NORM then
            if a.s ≠ b.s then
                c ← (a.s=0) ? G: L
            else
                if a.e > b.e then
                    af ← a.f
                    bf ← b.f || 0^(a.e-b.e)
                else
                    af ← a.f || 0^(b.e-a.e)
                    bf ← b.f
                endif
                if af = bf then
                    c ← E
                else
                    c ← ((a.s=0) ^ (af > bf)) ? G : L
                endif
            endif
        elseif a.t=NORM then
            c ← (a.s=0) ? G: L
        elseif b.t=NORM then
            c ← (b.s=0) ? G: L
        elseif a.t=ZERO and b.t=ZERO then
            c ← E
        else
            assert FALSE // should have covered al the cases above
        endif
enddef

def c ← fmul(a,b) as
        if a.t=NORM and b.t=NORM then
            c.s ← a.s ^ b.s
            c.t ← NORM
            c.e ← a.e + b.e
            c.f ← a.f * b.f
        // priority is given to b operand for NaN propagation
        elseif (b.t=SNAN) or (b.t=QNAN) then
            c.s ← a.s ^ b.s
            c.t ← b.t
            c.e ← b.e
            c.f ← b.f
        elseif (a.t=SNAN) or (a.t=QNAN) then
            c.s ← a.s ^ b.s
            c.t ← a.t
            c.e ← a.e
            c.f ← a.f
        elseif a.t=ZERO and b.t=INFINITY then
            c ← DEFAULTSNAN // Invalid
        elseif a.t=INFINITY and b.t=ZERO then
            c ← DEFAULTSNAN // Invalid
```

Fig. 29 (cont'd)

```
    elseif a.t=ZERO or b.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def c ← fdivr(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e - b.e + 256
        c.f ← (a.f || 0^256) / b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    elseif a.t=INFINITY then
        c.s ← a.s ^ b.s
        c.t ← INFINITY
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def msb ← findmsb(a) as
    MAXF ← 2^18 // Largest possible f value after matrix multiply
    for j ← 0 to MAXF
        if a_MAXF-1..j = (0^MAXF-1-j || 1) then
            msb ← j
        endif
    endfor
enddef

def ai ← PackF(prec,a,round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rn ← msb-1-fbits(prec) // lsb for normal
            rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
            rb ← (rn > rdn) ? rn : rdn
```

Fig. 29 (cont'd)

```
        if rb ≤ 0 then
            aifr ← a.f_{msb-1..0} || 0^{-rb}
            eadj ← 0
        else
            case round of
                C:
                    s ← 0^{msb-rb} || (~a.s)^{rb}
                F:
                    s ← 0^{msb-rb} || (a.s)^{rb}
                N, NONE:
                    s ← 0^{msb-rb} || ~a.f_{rb} || a.f_{rb}^{rb-1}
                X:
                    if a.f_{rb-1..0} ≠ 0 then
                        raise FloatingPointArithmetic // Inexact
                    endif
                    s ← 0
                Z:
                    s ← 0
            endcase
            v ← (0||a.f_{msb..0}) + (0||s)
            if v_{msb} = 1 then
                aifr ← v_{msb-1..rb}
                eadj ← 0
            else
                aifr ← 0^{fbits(prec)}
                eadj ← 1
            endif
        endif
        aien ← a.e + msb – 1 + eadj + ebias(prec)
        if aien ≤ 0 then
            if round = NONE then
                ai ← a.s || 0^{ebits(prec)} || aifr
            else
                raise FloatingPointArithmetic //Underflow
            endif
        elseif aien ≥ 1^{ebits(prec)} then
            if round = NONE then
                //default: round-to-nearest overflow handling
                ai ← a.s || 1^{ebits(prec)} || 0^{fbits(prec)}
            else
                raise FloatingPointArithmetic //Underflow
            endif
        else
            ai ← a.s || aien_{ebits(prec)-1..0} || aifr
        endif
    SNAN:
        if round ≠ NONE then
            raise FloatingPointArithmetic //Invalid
        endif
        if –a.e < fbits(prec) then
            ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..0} || 0^{fbits(prec)+a.e}
```

Fig. 29 (cont'd)

```
            else
                lsb ← a.f_{-a.e-1-fbits(prec)+1..0} ≠ 0
                ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..-a.e-1-fbits(prec)+2} || lsb
            endif
        QNAN:
            if -a.e < fbits(prec) then
                ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..0} || 0^{fbits(prec)+a.e}
            else
                lsb ← a.f_{-a.e-1-fbits(prec)+1..0} ≠ 0
                ai ← a.s || 1^{ebits(prec)} || a.f_{-a.e-1..-a.e-1-fbits(prec)+2} || lsb
            endif
        ZERO:
            ai ← a.s || 0^{ebits(prec)} || 0^{fbits(prec)}
        INFINITY:
            ai ← a.s || 1^{ebits(prec)} || 0^{fbits(prec)}
    endcase
defdef

def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rb ← -a.e
            if rb ≤ 0 then
                aifr ← a.f_{msb..0} || 0^{-rb}
                aims ← msb - rb
            else
                case round of
                    C, C.D:
                        s ← 0^{msb-rb} || (~ai.s)^{rb}
                    F, F.D:
                        s ← 0^{msb-rb} || (ai.s)^{rb}
                    N, NONE:
                        s ← 0^{msb-rb} || ~ai.f_{rb} || ai.f_{rb}^{rb-1}
                    X:
                        if ai.f_{rb-1..0} ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z, Z.D:
                        s ← 0
                endcase
                v ← (0||a.f_{msb..0}) + (0||s)
                if v_{msb} = 1 then
                    aims ← msb + 1 - rb
                else
                    aims ← msb - rb
                endif
                aifr ← v_{aims..rb}
            endif
            if aims > prec then
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (~as)^{prec-1}
```

Fig. 29 (cont'd)

```
                        C, F, N, X, Z:
                            raise FloatingPointArithmetic // Overflow
                    endcase
                elseif a.s = 0 then
                    ai ← aifr
                else
                    ai ← -aifr
                endif
            ZERO:
                ai ← 0^prec
            SNAN, QNAN:
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← 0^prec
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Invalid
                endcase
            INFINITY:
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (~as)^prec-1
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Invalid
                endcase
        endcase
enddef

def c ← frecrest(a) as
        b.s ← 0
        b.t ← NORM
        b.e ← 0
        b.f ← 1
        c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
        b.s ← 0
        b.t ← NORM
        b.e ← 0
        b.f ← 1
        c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
        if (a.t=NORM) then
            msb ← findmsb(a.f)
            a.e ← a.e + msb - 13
            a.f ← a.f_msb..msb-12 || 1
        else
            c ← a
        endif
enddef

def c ← fsqr(a) as
        if (a.t=NORM) and (a.s=0) then
            c.s ← 0
            c.t ← NORM
            if (a.e_0 = 1) then
```

Fig. 29 (cont'd)

```
                c.e ← (a.e-127) / 2
                c.f ← sqr(a.f || 0^127)
            else
                c.e ← (a.e-128) / 2
                c.f ← sqr(a.f || 0^128)
            endif
        elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
            c ← a
        elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
            c ← DEFAULTSNAN // Invalid
        else
            assert FALSE // should have covered al the cases above
        endif
enddef
```

Fig. 29 (cont'd)

Operation codes

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double

Fig. 30A

E.DIV.F.64.C	Ensemble divide floating-point double ceiling
E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

Fig. 30A (cont'd)

Selection

class	op	prec				round/trap
add	EADDF	16	32	64	128	NONE C F N X Z
divide	EDIVF	16	32	64	128	NONE C F N X Z
multiply	EMULF	16	32	64	128	NONE C F N X Z
complex multiply	EMUL.CF	16	32	64		NONE

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)

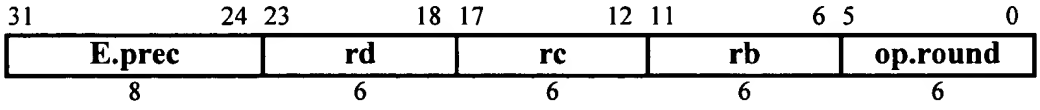


Fig. 30B

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,v_{size-1+i..i}),F(size,w_{size-1+j..j}))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,c_{i+prec-1..i})
        bi ← F(prec,b_{i+prec-1..i})
        case op of
            E.ADD.F:
                ai ← faddr(ci,bi,round)
            E.MUL.F:
                ai ← fmul(ci,bi)
            E.MUL.C.F:
                if (i and prec) then
                    ai ← fadd(mul(prec,c,i,b,i-prec), mul(prec,c,i-prec,b,i))
                else
                    ai ← fsub(mul(prec,c,I,b,I), mul(prec,c,i+prec,b,i+prec))
                endif
            E.DIV.F.:
                ai ← fdiv(ci,bi)
        endcase
        a_{i+prec-1..i} ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Fig. 30C

Operation codes

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

Fig. 31A

Selection

class	op	prec	round/trap
set	SET. E LG L GE	16 32 64 128	NONE X
subtract	SUB	16 32 64 128	NONE C F N X Z

Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)

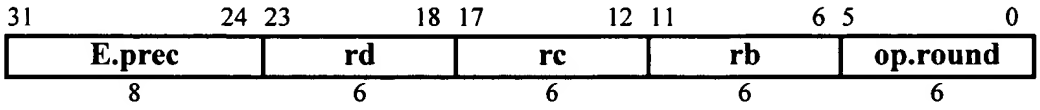


Fig. 31B

Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,c_{i+prec-1..i})
        bi ← F(prec,b_{i+prec-1..i})
        ai ← frsubr(ci,-bi, round)
        a_{i+prec-1..i} ← PackF(prec, ai, round)
    endfor
    RegWrite(rd, 128, a)
enddef
```

Fig. 31C

Operation codes

X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets

Fig. 32A

X.ROTR.32	Crossbar rotate right quadlets
X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

Fig. 32A (cont'd)

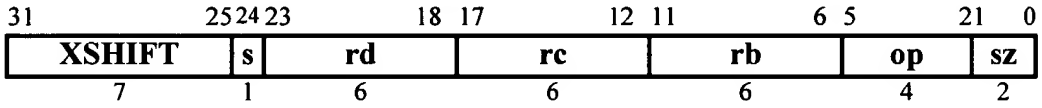
Selection

class	op	size
precision	EXPAND EXPAND.U COMPRESS COMPRESS.U	2 4 8 16 32 64 128
shift	ROTR ROTL SHR SHL SHL.O SHL.U.O SHR.U	2 4 8 16 32 64 128

Format

X.op.size rd=rc,rb

rd=xopsize(rc,rb)



lsize ← log(size)
s ← $lsize_2$
sz ← $lsize_{1..0}$

Fig. 32B

D. Appendix

In order to aid the examination of the present application, Applicant has provided two copies of the Appendix (already in the file in microfiche) in CD-ROM form in accordance with 37 C.F.R. § 1.52(e) and MPEP § 608.05. This amendment to the substitute specification is simply to aid the Examiner and the public in locating the already incorporated Appendix. No new matter has been added.

III. Double Patenting

Claims 1-17 stand rejected under 35 U.S.C. § 101 under the judicially created doctrine of obvious-type double patenting over claims 1-12 of a parent patent U.S. 6,295,599.

Although Applicants traverse the conclusion of obviousness under claims 1-12 of U.S. Patent 6,295,599 stated in the Office action, Applicants have filed a terminal disclaimer with this Amendment to render the issue moot.

IV. Prior Art Rejections

Claims 1, 7, 13, 15 and 16 stand rejected under 35 U.S.C. § 103 as being unpatentable over Gahan et al. (U.S. Patent No. 5,600,814) (hereafter "Gahan") in view of Kwon (U.S. Patent No. 5,768,546) (hereafter "Kwon").

Applicants respectfully traverse the rejection.

Obviousness can only be established by combining or modifying the teachings of the prior art to produce the ***claimed invention*** where there is some teaching, suggestion, or motivation to do so found either in the references themselves or in the knowledge generally available to one of ordinary skill in the art. Ecolochem Inc. v. Southern California Edison Co.,

B

227 F.3rd 1361, 56 U.S.P.Q.2d (BNA) 1065 (Fed. Cir. 2000); In re Dembiczak, 175 F.3d 994, 999, 50 U.S.P.Q.2D (BNA) 1614, 1617 (Fed. Cir. 1999); In re Fine, 837 F.2d 1071, 5 U.S.P.Q.2d 1596 (Fed. Cir. 1988); In re Jones, 958 F.2d 347, 21 U.S.P.Q.2d 1941 (Fed. Cir. 1992). See also MPEP 2143.01.

Neither Gahan nor Kwon, taken alone or in combination, disclose or suggest the invention recited in independent claims 1, 7, 13 and 15, nor dependent claim 16.

In reviewing the cited portions of Gahan and Kwon, it becomes apparent that Gahan and Kwon have been generalized, and, in fact, do not support the position asserted by the Examiner.

A system having . . . a *second memory system* having a data path width which is greater than the functional unit data path width and greater than the first data path width

In particular, neither Gahan nor Kwon, alone or in combination, teach or suggest the claim feature of "a system having . . . a *second memory system* having a data path width which is greater than the functional unit data path width and greater than the first data path width," as required by claim 1. In particular, Gahan teaches "[r]eferring to FIG. 1, [a] . . . system [which] comprises a [*single*] *main memory* coupled to a data bus 11 and a address bus 12." (See Gahan, col. 3, lines 4-5.) Therefore, Gahan does not teach a system which includes a *second memory system*, and, therefore, Gahan cannot teach a "*second memory system* having a data path width which is greater than the functional unit data path width and greater than the first data path width," as required by claim 1. Kwon does not teach "a system having a data path functional unit having a functional unit data path width, a first memory system having a first data path width, and a *second memory system* having a data path width which is greater than the functional unit data path width and greater than the first data path width," as required by claim 1. Therefore, neither Gahan nor Kwon, alone or in combination, can teach or suggest the required claim 1

B

element of "a system having . . . a *second memory system* having a data path width which is greater than the functional unit data path width and greater than the first data path width."

Copying a *second* memory operand portion from the first memory system to the second memory system, the *second* memory operand portion having the first data path width and *being catenated in the second memory system* with the first memory operand portion, *thereby forming catenated data*

In addition, neither Gahan nor Kwon, alone or in combination, teach or suggest the claim feature of "copying a *second* memory operand portion from the first memory system to the second memory system, the *second* memory operand portion having the first data path width and *being catenated in the second memory system* with the first memory operand portion, *thereby forming catenated data*," as required by claim 1. In particular, the Examiner admitted that Gahan does not teach this claim element. (See Office Action, page 3, paragraph 4.) In addition, Kwon teaches "a first latch for latching the *data word* from the first memory in a first predetermined number of byte; a first selector for selecting the bytes latched in the first latch in a second predetermined number of bytes." (See Kwon, col. 2, lines 56-59.) In other words, Kwon teaches (1) manipulating a *single* data word first with a latch and then (2) manipulating the *single* data word with a selector. Therefore, Kwon does not teach manipulating *two* memory operand portions or *two* data words, as required by claim 1. Therefore, Kwon cannot teach "copying a *second* memory operand portion from the first memory system to the second memory system," as required by claim 1.

In addition, Kwon teaches "*converting* a width of data word from the first system bus into a width of a data word of the second system bus, and outputting the converted data word to the second system bus." (See Kwon, col. 3, lines 19-22.) The *converting* which Kwon teaches is not the same as the *catenating* required by claim 1. In other words, Kwon does not teach *catenating* a *second* memory operand portion with a *first* memory operand portion and does not

B

teach *forming catenated data*, as required by claim 1. In addition, as discussed above, Kwon teaches manipulating a *single* data word with a *single memory system*. Therefore, Kwon cannot teach *catenating* a *second* memory operand portion in a *second memory system* with a *first* memory operand portion and cannot teach *forming catenated data*, as required by claim 1. Therefore, Kwon cannot teach "the *second* memory operand portion having the first data path and *being catenated in the second memory system* with the *first* memory operand portion, *thereby forming catenated data*," as required by claim 1.

Therefore, neither Gahan nor Kwon, alone or in combination, can teach or suggest the required claim element of "copying a *second* memory operand portion from the first memory system to the second memory system, the *second* memory operand portion having the first data path width and *being catenated in the second memory system* with the *first* memory operand portion, thereby *forming catenated data*," as required by claim 1.

Accordingly, Applicants respectfully request the above rejections to be withdrawn.

V. Conclusion

Applicant respectfully submits that the application is in condition for allowance, an early indication thereof is respectfully solicited. Should the Examiner have any questions or concerns regarding the amendments presented herein, the Examiner is invited to contact the undersigned representative of the Applicant.

To the extent necessary, a petition for an extension of time under 37 C.F.R. 1.136 is hereby made. Please charge any shortage in fees due in connection with the filing of this paper, including extension of time fees, to Deposit Account 500417 and please credit any excess fees to such deposit account.

Respectfully submitted,

MCDERMOTT, WILL & EMERY

Lawrence Cullen
Registration No. 44,489

600 13th Street, N.W.
Washington, DC 20005-3096
(202)756-8000
Facsimile: (202)756-8087
Date: March 24, 2003

B

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION

FOR

UNITED STATES PATENT

FOR

SYSTEM WITH WIDE OPERAND ARCHITECTURE, AND METHOD

INVENTORS:

Craig Hansen

John Moussouris

SPECIFICATION

APPLICATION

FOR

UNITED STATES PATENT

FOR

PROGRAMMABLE PROCESSOR WITH GROUP FLOATING-POINT OPERATIONS

INVENTORS:

Craig Hansen
Los Altos, California

John Moussouris
Palo Alto, California

SUBSTITUTE SPECIFICATION

~~———~~Related Applications

This application is a continuation of U.S. Patent Application No. 09/382,402, filed August 24, 1999, now U.S. Patent No. 6,295,599, which is a ~~continuation in part~~continuation-in-part of U.S. Patent Application No. 09/169,963, filed October 13, ~~1998 which is in turn related to~~1998, now U.S. Patent No. 6,006,318, which is a continuation of U.S. Patent Application No. ~~08/516,036, filed August 16,1995,~~08/754,827, filed November 22, 1996 now U.S. Patent No. ~~5,742,840.~~ 5,822,603, which is a divisional of U.S. Patent Application No. 08/516,036, filed August 16, 1995 now U.S. Patent No. 5,742,840.

Microfiche Appendix

Include herewith as an Appendix are 5 sheets of microfiche of 63 frames each.

Field of the Invention

The present invention relates to general purpose processor architectures, and particularly relates to wide operand architectures.

REFERENCE TO A "SEQUENCE LISTING," A TABLE, OR A COMPUTER PROGRAM LISTING APPENDIX SUBMITTED ON A COMPACT DISK

This application includes an appendix, submitted herewith in duplicate on compact disks labeled as "Copy 1" and "Copy 2." The contents of the compact disks are hereby incorporated by reference.

BACKGROUND OF THE INVENTION

The performance level of a processor, and particularly a general purpose processor, can be estimated from the multiple of a plurality of interdependent factors: clock rate, gates per clock, number of operands, operand and data path width, and operand and data path partitioning. Clock rate is largely influenced by the choice of circuit and logic technology, but is also influenced by the number of gates per clock. Gates per clock is how many gates in a pipeline may change state in a single clock cycle. This can be reduced by inserting latches into the data path: when the number of gates between latches is reduced, a higher clock is possible. However, the additional latches produce a longer pipeline length, and thus come at a cost of increased instruction latency. The number of operands is straightforward; for example, by adding with carry-save techniques, three values may be added together with little more delay than is required for adding two values. Operand and data path width defines how much data can be processed at once; wider data paths can perform more complex functions, but generally this comes at a higher implementation cost. Operand and data path partitioning refers to the efficient use of the data path as width is increased, with the objective of maintaining substantially peak usage.

The last factor, operand and data path partitioning, is treated extensively in commonly-assigned U.S. Patent No.'s 5,742,840, 5,794,060, 5,794,061, 5,809,321, and 5,822,603, which describe systems and methods for enhancing the utilization of a general purpose processor by adding classes of instructions. These classes of instructions use the contents of general purpose registers as data path sources, partition the operands into symbols of a specified size, perform operations in parallel, catenate the results and place the catenated results into a general-purpose register. These patents, all of which are

assigned to the same assignee as the present invention, teach a general purpose microprocessor which has been optimized for processing and transmitting media data streams through significant parallelism.

While the foregoing patents offered significant improvements in utilization and performance of a general purpose microprocessor, particularly for handling broadband communications such as media data streams, other improvements are ~~possible.~~ possible.

Many general purpose processors have general registers to store operands for instructions, with the register width matched to the size of the data path. Processor designs generally limit the number of accessible registers per instruction because the hardware to access these registers is relatively expensive in power and area. While the number of accessible registers varies among processor designs, it is often limited to two, three or four registers per instruction when such instructions are designed to operate in a single processor clock cycle or a single pipeline flow. Some processors, such as the Motorola 68000 have instructions to save and restore an unlimited number of registers, but require multiple cycles to perform such an instruction.

The Motorola 68000 also attempts to overcome a narrow data path combined with a narrow register file by taking multiple cycles or pipeline flows to perform an instruction, and thus emulating a wider data path. However, such multiple precision techniques offer only marginal improvement in view of the additional clock cycles required. The width and accessible number of the general purpose registers thus fundamentally limits the amount of processing that can be performed by a single instruction in a register-based machine.

Existing processors may provide instructions that accept operands for which one or more operands are read from a general purpose processor's memory system. However, as these memory operands are generally specified by register operands, and the memory system data path is no wider than the processor data path, the width and accessible number of general purpose operands per instruction per cycle or pipeline flow is not enhanced.

The number of general purpose register operands accessible per instruction is generally limited by logical complexity and instruction size. For example, it might be possible to implement certain desirable but complex functions by specifying a large number of general purpose registers, but substantial additional logic would have to be added to a conventional design to permit simultaneous reading and bypassing of

the register values. While dedicated registers have been used in some prior art designs to increase the number or size of source operands or results, explicit instructions load or store values into these dedicated registers, and additional instructions are required to save and restore these registers upon a change of processor context.

There has therefore been a need for a processor system capable of efficient handling of operands of greater width than either the memory system or any accessible general purpose register.

SUMMARY OF THE INVENTION

The present invention provides a system and method for improving the performance of general purpose processors by expanding at least one source operand to a width greater than the width of either the general purpose register or the data path width. In addition, several classes of instructions will be provided which cannot be performed efficiently if the operands are limited to the width and accessible number of general purpose registers.

In the present invention, operands are provided which are substantially larger than the data path width of the processor. This is achieved, in part, by using a general purpose register to specify a memory address from which at least more than one, but typically several data path widths of data can be read. To permit such a wide operand to be performed in a single cycle, the data path functional unit is augmented with dedicated storage to which the memory operand is copied on an initial execution of the instruction. Further execution of the instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value. However, such reads are subject to conditions to verify that the memory operand has not been altered by intervening instructions. If the memory operand remains current – that is, the conditions are met – the memory operand fetch can be combined with one or more register operands in the functional unit, producing a result. The size of the result is, typically, constrained to that of a general register so that no dedicated or other special storage is required for the result.

Exemplary instructions using wide operations include wide instructions that

perform bit-levelbit level switching (Wide Switch), byte or larger table-lookup (Wide Translate), Wide Multiply Matrix, Wide Multiply Matrix Extract, Wide Multiply Matrix Extract Immediate, Wide Multiply Matrix Floating point, and Wide Multiply Matrix Galois.

Another aspect of the present invention addresses efficient usage of a multiplier array that is fully used for high precision arithmetic, but is only partly used for other, lower precision operations. This can be accomplished by extracting the high-order portion of the multiplier product or sum of products, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion. The rounding may be any of several types, including round-to-nearest/even, toward zero, floor, or ceiling. Overflows are typically handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

When an extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled for use in subsequent operations without concern of overflow or rounding. As a result, performance is enhanced. In those instances where the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing such control information in a single register, the size of the instruction is reduced over the number of bits that such an instruction would otherwise require, again improving performance and enhancing processor flexibility. Exemplary instructions are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract, and Ensemble Scale Add Extract. With particular regard to the Ensemble Scale Add Extract Instruction, the extract control information is combined in a register with two values used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers otherwise required, thus reducing the number of bits required for the instruction.

THE FIGURES

Figure 1 is a system level diagram showing the functional blocks of a system

~~according to~~in accordance with an exemplary embodiment of the present invention.

Figure 2 is a matrix representation of a wide matrix multiply in accordance with an exemplary embodiment of the present invention.

Figure 3 is a further representation of a wide matrix multiple in accordance with an exemplary embodiment of the present invention.

Figure 4 is a system level diagram showing the functional blocks of a system incorporating a combined Simultaneous Multi Threading and Decoupled Access from Execution processor in accordance with an exemplary embodiment of the present invention.

Figure 5 illustrates a wide operand in accordance with an exemplary embodiment of the present invention.

Figure 6 illustrates an approach to specifier decoding in accordance with an exemplary embodiment of the present invention.

Figure 7 illustrates in operational block form a Wide Function Unit in accordance ~~with~~ with an exemplary embodiment of the present invention.

Figure 8 illustrates in flow diagram form the Wide Microcache control function in accordance with an exemplary embodiment of the present invention.

Figure 9 illustrates Wide Microcache data structures in accordance with an exemplary embodiment of the present invention.

Figures 10 and 11 illustrate a Wide Microcache control in accordance with an exemplary embodiment of the present invention.

Figures 12A-12D illustrate a Wide Switch instruction in accordance with an exemplary embodiment of the present invention.

Figures 13A-13D illustrate a Wide Translate instruction in accordance with an exemplary embodiment of the present invention.

Figures 14A - 14E illustrate a Wide Multiply Matrix instruction in accordance with an exemplary embodiment of the present invention.

Figures 15A - 15F illustrate a Wide Multiply Matrix Extract instruction in accordance with an exemplary embodiment of the present invention.

Figures 16A - 16E illustrate a Wide Multiply Matrix Extract Immediate instruction in accordance with an exemplary embodiment of the present invention.

Figures 17A - 17E illustrate a Wide Multiply Matrix Floating point instruction in accordance with an exemplary embodiment of the present invention.

Figures 18A - 18D illustrate a Wide Multiply Matrix Galois instruction in accordance with an exemplary embodiment of the present invention.

Figures 19A - 19G illustrate an Ensemble Extract Inplace instruction in accordance with an exemplary embodiment of the present invention.

Figures 20A - 20J illustrate an Ensemble Extract instruction in accordance with an exemplary embodiment of the present invention.

Figures 21A - 21B illustrate a System and Privileged Library Calls in accordance with an exemplary embodiment of the present invention.

Figures 22A - 22B illustrate an Ensemble Scale-Add Floating-point instruction in accordance with an exemplary embodiment of the present invention.

Figures 23A - 23C illustrate a Group Boolean instruction in accordance with an exemplary embodiment of the present invention.

Figures 24A - 24C illustrate a Branch Hint instruction in accordance with an exemplary embodiment of the present invention.

Figures 25A - 25C illustrate an Ensemble Sink Floating-point instruction in accordance with an exemplary embodiment of the present invention.

Figures 26A - 26C illustrate Group Add instructions in accordance with an exemplary embodiment of the present invention.

Figures 27A - 27C illustrate Group Set instructions and Group Subtract instructions in accordance with an exemplary embodiment of the present invention.

Figures 28A - 28C illustrate Ensemble Convolve, Ensemble Divide, Ensemble Multiply, and Ensemble Multiply Sum instructions in accordance with an exemplary embodiment of the present invention.

Figure 29 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections.

Figures 30A - 30C illustrate Ensemble Floating-Point Add, Ensemble Floating-Point Divide, and Ensemble Floating-Point Multiply instructions in accordance with an exemplary embodiment of the present invention.

Figures 31A - 31C illustrate Ensemble Floating-Point Subtract instructions in

accordance with an exemplary embodiment of the present invention.

Figures 32A - 32D illustrate Crossbar Compress, Expand, Rotate, and Shift instructions in accordance with an exemplary embodiment of the present invention.

Figures 33A - 33D illustrate Extract instructions in accordance with an exemplary embodiment of the present invention.

Figures 34A - 34E illustrate Shuffle instructions in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Processor Layout

Referring first to Figure 1, a general purpose processor is illustrated therein in block diagram form. In Figure 1, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 101-104. Each access instruction fetch queue A-Queue ~~101-104~~101-104 is coupled to an access register file AR 105-108, which are each coupled to two access functional units A 109-116. In a typical embodiment, each thread of the processor may have on the order of sixty-four general purpose registers (e.g., the ~~AR's~~AR's 105-108 and ~~ER's~~ER's 125-128). The access units function independently for four simultaneous threads of execution, and each compute program control flow by performing arithmetic and branch instructions and access memory by performing load and store instructions. These access units also provide wide operand specifiers for wide operand instructions. These eight access functional units A 109-116 produce results for access register files AR~~105-~~ 105-108 and memory addresses to a shared memory system 117-120.

~~The~~In one embodiment, the memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. In Fig. 1, the memory system is comprised of a combined cache and niche memory 117, an external bus interface 118, and, externally to the device, a secondary cache 119 and main memory system with I/O devices 120. The memory contents fetched from memory system 117-120 are combined with execute instructions not performed by the access unit, and entered into the four execute instruction queues E-Queue 121-124. For wide instructions, memory contents fetched from memory system 117-120 are also provided to wide operand microcaches 132-136 by bus 137. Instructions and memory data

from E-queue 121-124 are presented to execution register files 125-128, which fetch execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 131, that selects which instructions from the four threads are to be routed to the available execution functional units E 141 and 149, X 142 and 148, G 143-144 and 146-147, and T 145. The execution functional units E 141 and 149, the execution functional units X 142 and 148, and the execution functional unit T 145 each contain a wide operand microcache 132-136, which are each coupled to the memory system 117 by bus 137.

The execution functional units G 143-144 and 146-147 are group arithmetic and logical units that perform simple arithmetic and logical instructions, including group operations wherein the source and result operands represent a group of values of a specified symbol size, which are partitioned and operated on separately, with results catenated together. In a presently preferred embodiment the data path is 128 bits wide, although the present invention is not intended to be limited to any specific size of data path.

The execution functional units X 142 and 148 are crossbar switch units that perform crossbar switch instructions. The crossbar switch units 142 and 148 perform data handling operations on the data stream provided over the data path source operand buses 151-158, including deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, plus the wide operations discussed hereinafter. In a key element of a first aspect of the invention, at least one such operation will be expanded to a width greater than the general register and data path ~~width. Examples of the data manipulation operations are described in the System~~ width.

~~Architecture/BroadMX Architecture descriptions included herein.~~

The execution functional units E 141 and 149 are ensemble units that perform ensemble instructions using a large array multiplier, including group or vector multiply and matrix multiply of operands partitioned from data path source operand buses 151-158 and treated as integer, ~~floating point,~~floating point, polynomial or Galois field values. Matrix multiply instructions and other operations~~described in the System Architecture/BroadMX Architecture descriptions included herein~~ utilize a wide operand loaded into the wide operand microcache 132 and 136.

The execution functional unit T 145 is a translate unit that performs table-look-up

operations on a group of operands partitioned from a register operand, and catenates the result. The Wide Translate instruction~~, described in the System Architecture/BroadMX Architecture descriptions included herein,~~ utilizes a wide operand loaded into the wide operand microcache 134.

The execution functional units E 141, 149, execution functional units X ~~142,~~ 142, 148, and execution functional unit T each contain dedicated storage to permit storage of source operands including wide operands as discussed hereinafter. The dedicated storage ~~132-136,~~132 - 136, which may be thought of as a wide microcache, typically has a width which is a multiple of the width of the data path operands related to the data path source operand buses 151-158. Thus, if the width of the data path ~~151-158~~151 - 158 is 128 bits, the dedicated storage ~~132-136~~132 - 136 may have a width of 256, 512, 1024 or 2048 bits. Operands which utilize the full width of the dedicated storage are referred to herein as wide operands, although it is not necessary in all instances that a wide operand use the entirety of the width of the dedicated storage; it is sufficient that the wide operand use a portion greater than the width of the memory data path of the output of the memory system 117-120 and the functional unit data path of the input of the execution functional units 141-149, though not necessarily greater than the width of the two combined. Because the width of the dedicated storage 132-136 is greater than the width of the memory operand bus 137, portions of wide operands are loaded sequentially into the dedicated storage 132-136. However, once loaded, the wide operands may then be used at substantially the same time. It can be seen that functional units 141-149 and associated execution registers 125-128 form a data functional unit, the exact elements of which may vary with implementation.

The execution register file ER 125-128 source operands are coupled to the execution units 141-145 using source operand buses 151-154 and to the execution units 145-149 using source operand buses 155-158. The function unit result operands from execution units 141-145 are coupled to the execution register file ER 125-128 using result bus 161 and the function units result operands from execution units 145-149 are coupled to the execution register file using result bus ~~162.~~
162.

Wide Multiply Matrix

The wide operands of the present invention provide the ability to execute complex instructions such as the wide multiply matrix instruction shown in Figure 2, which can be appreciated in an alternative form, as well, from Figure 3. As can be appreciated from Figures 2 and 3, a wide operand permits, for example, the matrix multiplication of various sizes and shapes which exceed the data path width. The example of Figure 2 involves a matrix specified by register rc havinga 128*64/size bits (512 bits for this example) multiplied by a vector contained in register rb having ~~a 128 size,~~128 bits, to yield a result, placed in register rd, of 128 bits.

The notation used in Figure 2 and following similar figures illustrates a multiplication as a shaded area at the intersection of two operands projected in the horizontal and vertical dimensions. A summing node is illustrated as a line segment connecting a darkened dots at the location of multiplier products that are summed. Products that are subtracted at the summing node are indicated with a minus symbol within the shaded area.

When the instruction operates on floating-point values, the multiplications and summations illustrated are floating point multiplications and summations. An exemplary embodiment may perform these operations without rounding the intermediate results, thus computing the final result as if computed to infinite precision and then rounded only once.

It can be appreciated that an exemplary embodiment of the multipliers may compute the product in carry-save form and may encode the multiplier rb using Booth encoding to minimize circuit area and delay. It can be appreciated that an exemplary embodiment of such summing nodes may perform the summation of the products in any order, with particular attention to minimizing computation delay, such as by performing the additions in a binary or higher-radix tree, and may use carry-save adders to perform the addition to minimize the summation delay. It can also be appreciated that an exemplary embodiment may perform the summation using sufficient intermediate precision that no fixed-point or floating-point overflows occur on intermediate results.

A comparison of Figures 2 and 3 can be used to clarify the relation between the notation used in Figure 2 and the more conventional schematic notation in Figure 3, as the same operation is illustrated in these two figures.

Wide Operand

The operands that are substantially larger than the data path width of the processor are provided by using a general-purpose register to specify a memory specifier from which more than one but in some embodiments several data path widths of data can be read into the dedicated storage. The memory specifier typically includes the memory address together with the size and shape of the matrix of data being operated on. The memory specifier or wide operand specifier can be better appreciated from Figure 5, in which a specifier 500 is seen to be an address, plus a field representative of the size/2 and a further field representative of width/2, where size is the product of the depth and width of the data. The address is aligned to a specified size, for example ~~sixty-four~~sixty four bytes, so that a plurality of low order bits (for example, six bits) are zero. The specifier 500 can thus be seen to comprise a first field 505 for the address, plus two field indicia 510 within the low order six bits to indicate size and width.

Specifier Decoding

The decoding of the specifier 500 may be further appreciated from Figure 6 where, for a given specifier 600 made up of an address field 605 together with a field 610 comprising plurality of low order bits. By a series of arithmetic operations shown at steps 615 and 620, the portion of the field 610 representative of width/2 is developed. In a similar series of steps shown at 625 and 630, the value of t is decoded, which can then be used to decode both size and address. The portion of the field 610 representative of size/2 is decoded as shown at steps 635 and 640, while the address is decoded in a similar way at steps 645 and 650.

Wide Function Unit

The wide function unit may be better appreciated from Figure 7, in which a register number 700 is provided to an operand checker 705. Wide operand specifier 710 communicates with the operand checker 705 and also addresses memory 715 having a defined memory width. The memory address includes a plurality of register operands ~~720A-n,~~720A n, which are accumulated in a dedicated storage portion 714 of a data functional unit 725. In the exemplary embodiment shown in Figure 7, the dedicated storage 71.4 can be seen to have a width equal to eight data path widths, such that eight wide operand portions 730A-H are sequentially loaded into the dedicated storage to form

the wide operand. Although eight portions are shown in Figure 7, the present invention is not limited to eight or any other specific multiple of data path widths. Once the wide operand portions 730A-H are sequentially loaded, they may be used as a single wide operand 735 by the functional element 740, which may be any element(s) from Figure 1 connected thereto. The result of the wide operand is then provided to a result register 745, which in a presently preferred embodiment is of the same width as the memory width.

Once the wide operand is successfully loaded into the dedicated storage 714, a second aspect of the present invention may be appreciated. Further execution of this instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value under specific conditions that determine whether the memory operand has been altered by intervening instructions. Assuming that these conditions are met, the memory operand fetch from the dedicated storage is combined with one or more register operands in the functional unit, producing a result. In some embodiments, the size of the result is limited to that of a general register, so that no similar dedicated storage is required for the result. However, in some different embodiments, the result may be a wide operand, to further enhance performance.

To permit the wide operand value to be addressed by subsequent instructions specifying the same memory address, various conditions must be checked and ~~confirmed:~~ confirmed:

Those conditions include:

1. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the storage to be marked invalid, since a memory store instruction directed to any of the memory addresses stored in dedicated storage 714 means that data has been ~~overwritten.~~ overwritten.

2. The register number used to address the storage is recorded. If no intervening instructions have written to the register, and the same register is used on the subsequent instruction, the storage is valid (unless marked invalid by rule #1).

3. If the register has been modified or a different register number is used, the value of the register is read and compared against the address recorded for the dedicated storage. This uses more resources than #1 because of the

need to fetch the register contents and because the width of the register is greater than that of the register number itself. If the address matches, the storage is valid. The new register number is recorded for the dedicated storage.

4. If conditions #2 or #3 are not met, the register contents are used to address the general-purpose ~~processor's~~processor's memory and load the dedicated storage. If dedicated storage is already fully loaded, a portion of the dedicated storage must be discarded (victimized) to make room for the new value. The instruction is then performed using the newly updated dedicated storage. The address and register number is recorded for the dedicated storage.

By checking the above conditions, the need for saving and restoring the dedicated storage is eliminated. In addition, if the context of the processor is changed and the new context does not employ Wide instructions that reference the same dedicated storage, when the original context is restored, the contents of the dedicated storage are allowed to be used without refreshing the value from memory, using checking rule #3. Because the values in the dedicated storage are read from memory and not modified directly by performing wide operations, the values can be discarded at any time without saving the results into general memory. This property simplifies the implementation of rule #4 above.

An alternate embodiment of the present invention can replace rule #1 above with the following rule:

1.a. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the dedicated storage to be updated, as well as the general memory.

By use of the above rule 1.a, memory store instructions can modify the dedicated storage, updating just the piece of the dedicated storage that has been changed, leaving the remainder intact. By continuing to update the general memory, it is still true that the contents of the dedicated memory can be discarded at any time without saving the results into general memory. Thus rule #4 is not made more complicated by this choice. The

advantage of this alternate embodiment is that the dedicated storage need not be discarded (invalidated) by memory store operations.

Wide Microcache Data Structures

Referring next to Figure 9, an exemplary arrangement of the data structures of the wide microcache or dedicated storage_114 may be better appreciated. The wide microcache contents, wmc.c, can be seen to form a plurality of data path widths 900A-n, although in the example shown the number is eight. The physical address, wmc.pa, is shown as 64 bits in the example shown, although the invention is not limited to a specific width. The size of the contents, wmc.size, is also provided in a field which is shown as 10 bits in an exemplary embodiment. A "contents valid" flag, wmc.cv, of one bit is also included in the data structure, together with a two bit field for thread last used, or wmc.th. In addition, a six bit field for register last used, wmc.reg, is provided in an exemplary embodiment. Further, a one bit flag for register and thread valid, or wmc.rtv, may be provided.

Wide Microcache Control - Software

The process by which the microcache is initially written with a wide operand, and thereafter verified as valid for fast subsequent operations, may be better appreciated from ~~Figure 8.~~Figure 8. The process begins at 800, and progresses to step 805 where a check of the register contents is made against the stored value wmc.rc. If true, a check is made at step 810 to verify the thread. If true, the process then advances to step 815 to verify whether the register and thread are valid. If step 815 reports as true, a check is made at step 820 to verify whether the contents are valid. If all of steps 805 through 820 return as true, the subsequent instruction is able to utilize the existing wide operand as shown at step 825, after which the process ends. However, if any of steps 805 through 820 return as false, the process branches to step 830, where content, physical address and size are set. Because steps 805 through 820 all lead to either step 825 or 830, steps 805 through 820 may be performed in any order or simultaneously without altering the process. The process then advances to step 835 where size is checked. This check basically ensures that the size of the translation unit is greater than or equal to the size of the wide operand, so

that a physical address can directly replace the use of a virtual address. The concern is that, in some embodiments, the wide operands may be larger than the minimum region that the virtual memory system is capable of mapping. As a result, it would be possible for a single contiguous virtual address range to be mapped into multiple, disjoint physical address ranges, complicating the task of comparing physical addresses. By determining the size of the wide operand and comparing that size against the size of the virtual address mapping region which is referenced, the instruction is aborted with an exception trap if the wide operand is larger than the mapping region. This ensures secure operation of the processor. Software can then re-map the region using a larger size map to continue execution if desired. Thus, if size is reported as unacceptable at step 835, an exception is generated at step 840. If size is acceptable, the process advances to step 845 where physical address is checked. If the check reports as met, the process advances to step 850, where a check of the contents valid flag is made. If either check at step 845 or 850 reports as false, the process branches and new content is written into the dedicated storage 114, with the fields thereof being set accordingly. Whether the check at step 850 reported true, or whether new content was written at step 855, the process advances to step 860 where appropriate fields are set to indicate the validity of the data, after which the requested function can be performed at step 825. The process then ends.

Wide Microcache Control - Hardware

Referring next to Figures 10 and 11, which together show the operation of the microcache controller from a hardware standpoint, the operation of the microcache controller may be better understood. In the hardware implementation, it is clear that conditions which are indicated as sequential steps in Figure 8 and 9 above can be performed in parallel, reducing the delay for such wide operand checking. Further, a copy of the indicated hardware may be included for each wide microcache, and thereby all such microcaches as may be alternatively referenced by an instruction can be tested in parallel. It is believed that no further discussion of Figures 10 and 11 is required in view of the extensive discussion of Figures 8 and 9, above.

Various alternatives to the foregoing approach do exist for the use of wide operands, including an implementation in which a single instruction can accept two wide

operands, partition the operands into symbols, multiply corresponding symbols together, and add the products to produce a single scalar value or a vector of partitioned values of width of the register file, possibly after extraction of a portion of the sums. Such an instruction can be valuable for detection of motion or estimation of motion in video compression. A further enhancement of such an instruction can incrementally update the dedicated storage if the address of one wide operand is within the range of previously specified wide operands in the dedicated storage, by loading only the portion not already within the range and shifting the in-range portion as required. Such an enhancement allows the operation to be performed over a ~~"sliding window"~~"sliding window" of possible values. In such an instruction, one wide operand is aligned and supplies the size and shape information, while the second wide operand, updated incrementally, is not aligned.

Another alternative embodiment of the present invention can define additional instructions where the result operand is a wide operand. Such an enhancement removes the limit that a result can be no larger than the size of a general register, further enhancing performance. These wide results can be cached locally to the functional unit that created them, but must be copied to the general memory system before the storage can be reused and before the virtual memory system alters the mapping of the address of the wide result. Data paths must be added so that load operations and other wide operations can read these wide results -- forwarding of a wide result from the output of a functional unit back to its input is relatively easy, but additional data paths may have to be introduced if it is desired to forward wide results back to other functional units as wide operands.

As previously discussed, a specification of the size and shape of the memory operand is included in the low-order bits of the address. In a presently preferred implementation, such memory operands are typically a power of two in size and aligned to that size. Generally, ~~one-half~~one half the total size is added (or inclusively ~~or'ed,~~or'ed, or exclusively ~~or'ed)~~or'ed) to the memory address, and one half of the data width is added (or inclusively ~~or'ed,~~or'ed, or exclusively ~~or'ed)~~or'ed) to the memory address. These bits can be decoded and stripped from the memory address, so that the controller is made to step through all the required addresses. This decreases the number of distinct operands required for these instructions, as the size, shape and address of the memory operand are combined into a single register operand value.

The following table illustrates the arithmetic and descriptive notation used in the pseudocode in the Figures referenced hereinafter:

x+y	two's complement addition of x and y. Result is the same size as the operands, and operands must be of equal size.
x - y	two's complement subtraction of y from x. Result is the same size as the operands, and operands must be of equal size.
x * y	two's complement multiplication of x and y. Result is the same size as the operands, and operands must be of equal size.
x / y	two's complement division of x by y. Result is the same size as the operands, and operands must be of equal size.
x & y	bitwise and of x and y. Result is same size as the operands, and operands must be of equal size.
x | y	bitwise or of x and y. Result is same size as the operands, and operands must be of equal size.
x ^ y	bitwise exclusive-of of x and y. Result is same size as the operands, and operands must be of equal size.
~x	bitwise inversion of x. Result is same size as the operand.
x = y	two's complement equality comparison between x and y. Result is a single bit, and operands must be of equal size.
x ≠ y	two's complement inequality comparison between x and y. Result is a single bit, and operands must be of equal size.
x < y	two's complement less than comparison between x and y. Result is a single bit, and operands must be of equal size.
x > y	two's complement greater than or equal comparison between x and y. Result is a single bit, and operands must be of equal size.
$\sqrt{}$	floating-point square root of x
x y	concatenation of bit field x to left of bit field y
$_x$y	binary digit x repeated, concatenated y times. Size of result is y.
x_y	extraction of bit y (using little-endian bit numbering) from value x. Result is a single bit.
$x_{y..z}$	extraction of bit field formed from bits y through z of value x. Size of result is -z+1; if z>y, result is an empty string.

x?y:z	value of y, if x is true, otherwise value of z. Value of x is a single bit.
x ←y	bitwise assignment of x to value of y
x.y	subfield of structured bitfield x
Sn	signed, two's complement, binary data format of n bytes
Un	unsigned binary data format of n bytes
Fn	floating-point data format of n bytes

Wide Operations

Particular examples of wide operations which are defined by the present invention include the Wide Switch instruction that performs bit-level switching; the Wide Translate instruction which performs byte (or larger) ~~table lookup;~~table lookup; Wide Multiply ~~Matrix,~~Matrix; Wide Multiply Matrix Extract and Wide Multiply Matrix Extract Immediate (discussed below), Wide Multiply Matrix Floating-point, and Wide Multiply Matrix Galois (also discussed below). While the discussion below focuses on particular sizes for the exemplary instructions, it will be appreciated that the invention is not limited to a particular width.

Wide Switch

~~The~~An exemplary embodiment of the Wide Switch instruction is shown in Figures 12A-12D. In an exemplary embodiment, the Wide Switch instruction rearranges the contents of up to two registers (256 bits) at the bit level, producing a full-width (128 bits) register result. To control the rearrangement, a wide operand specified by a single register, consisting of eight bits per bit position is used. For each result bit position, eight wide operand bits for each bit position select which of the 256 possible source register bits to place in the result. When a wide operand size smaller than 128 bytes is specified, the high order bits of the memory operand are replaced with values corresponding to the result bit position, so that the memory operand specifies a bit selection within symbols of the operand size, performing the same operation on each symbol.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results

together, placing the result in a general register. An exemplary embodiment of the format 1210 of the Wide Switch instruction is shown in Fig. 12A.

An exemplary embodiment of a schematic 1230 of the Wide Switch instruction is shown in Fig. 12B. In an exemplary embodiment, the contents of register rc specifies a virtual address apd optionally an operand size, and a value of specified size is loaded from memory. A second value is the catenated contents of registers rd and rb. Eight corresponding bits from the memory value are used to select a single result bit from the second value, for each corresponding bit position. The group of results is catenated and placed in register ra.

In an exemplary embodiment, the virtual address must either be aligned to 128 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes. An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be 8, 16, 32, 64, or 128 bytes. If the address is not valid an "access disallowed by virtual address" exception occurs. When a size smaller than 128 bits is specified, the high order bits of the memory operand are replaced with values corresponding to the bit position, so that the same memory operand specifies a bit selection within symbols of the operand size, and the same operation is performed on each symbol.

In an exemplary embodiment, a wide switch (W.SWITCH.L or W.SWITCH.B) instruction specifies an 8-bit location for each result bit from the memory operand, that selects one of the 256 bits represented by the catenated contents of registers rd and rb.

An exemplary embodiment of the pseudocode 1250 of the Wide Switch instruction is shown in Fig. 12C. An exemplary embodiment of the exceptions 1280 of the Wide Switch instruction is shown in Fig. 12D.

Wide Translate

An exemplary embodiment of the Wide Translate instruction is shown in Figures 13A-13D. In an exemplary embodiment, the Wide Translate instructions use a wide operand to specify a table of depth up to 256 entries and width of up to 128 bits. The contents of a register is partitioned into operands of one, two, four, or eight bytes, and the partitions are used to select values from the table in parallel. The depth and width of the

table can be selected by specifying the size and shape of the wide operand as described above.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1310 of the Wide Translate instruction is shown in Fig. 13A.

An exemplary embodiment of the schematic 1330 of the Wide Translate instruction is shown in Fig. 13B. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of a size specified. The low-order bytes of the second group of values are used as addresses to choose entries from one or more tables constructed from the first value, producing a group of values. The group of results is catenated and placed in register rd.

In an exemplary embodiment, by default, the total width of tables is 128 bits, and a total table width of 128, 64, 32, 16 or 8 bits, but not less than the group size may be specified by adding the desired total table width in bytes to the specified address: 16, 8, 4, 2, or 1. When fewer than 128 bits are specified, the tables repeat to fill the 128 bit width.

In an exemplary embodiment, the default depth of each table is 256 entries, or in bytes is 32 times the group size in bits. An operation may specify 4, 8, 16, 32, 64, 128 or 256 entry tables, by adding one half of the memory operand size to the address. Table index values are masked to ensure that only the specified portion of the table is used. Tables with just 2 entries cannot be specified; if 2-entry tables are desired, it is recommended to load the entries into registers and use G.MUX to select the table entries.

In an exemplary embodiment, failing to initialize the entire table is a potential security hole, as an instruction in with a small-depth table could access table entries previously initialized by an instruction with a large-depth table. This security hole may be closed either by initializing the entire table, even if extra cycles are required, or by masking the index bits so that only the initialized portion of the table is used. An exemplary embodiment may initialize the entire table with no penalty in cycles by writing to as many as 128 table entries at once. Initializing the entire table with writes to only one

entry at a time requires writing 256 cycles, even when the table is smaller. Masking the index bits is the preferred solution.

In an exemplary embodiment, masking the index bits suggests that this instruction, for tables larger than 256 entries, may be extended to a general-purpose memory translate function where the processor performs enough independent load operations to fill the 128 bits. Thus, the 16, 32, and 64 bit versions of this function perform equivalent of 8, 4, 2 withdraw, 8, 4, or 2 load-indexed and 7, 3, or 1 group-extract instructions. In other words, this instruction can be as powerful as 23, 11, or 5 previously existing instructions. The 8-bit version is a single cycle operation replacing 47 existing instructions, so these extensions are not as powerful, but nonetheless, this is at least a 50% improvement on a 2-issue processor, even with one cycle per load timing. To make this possible, the default table size would become 65536, 2^32 and 2^64 for 16, 32 and 64-bit versions of the instruction.

In an exemplary embodiment, for the big-endian version of this instruction, in the definition below, the contents of register rb is complemented. This reflects a desire to organize the table so that the lowest. addressed table entries are selected when the index is zero. In the logical implementation, complementing the index can be avoided by loading the table memory differently for big-endian and little-endian versions; specifically by loading the table into memory so that the highest-addressed table entries are selected when the index is zero for a big-endian version of the instruction. In an exemplary embodiment of the logical implementation, complementing the index can be avoided by loading the table memory differently for big-endian and little-endian versions. In order to avoid complementing the index, the table memory is loaded differently for big-endian versions of the instruction by complementing the addresses at which table entries are written into the table for a big-endian version of the instruction.

In an exemplary embodiment, the virtual address must either be aligned to 4096 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or the desired total table width in bytes. An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be a power of two from 4 to 4096 bytes, but must be at least 4 times the group size and 4 times the total table width. If the address is not valid an "access disallowed by virtual

address" exception occurs.

In an exemplary embodiment, a wide translate (W.TRANSLATE.8.L or W.TRANSLATE.8.B) instruction specifies a translation table of 16 entries (vsize=16) in depth, a group size of 1 byte (gsize=8 bits), and a width of 8 bytes (wsize=64 bits). The address specifies a total table size (msize=1024 bits=vsize*wsize) and a table width (wsize=64 bits) by adding one half of the size in bytes of the table (64) and adding the size in bytes of the table width (8) to the table address in the address specification. The instruction will create duplicates of this table in the upper and lower 64 bits of the data path, so that 128 bits of operand are processed at once, yielding a 128 bit result.

An exemplary embodiment of the pseudocode 1350 of the Wide Translate instruction is shown in Fig. 13C. An exemplary embodiment of the exceptions 1380 of the Wide Translate instruction is shown in Fig. 13D.

Wide Multiply Matrix

An exemplary embodiment of the Wide Multiply Matrix instruction is shown in Figures 14A-14E. In an exemplary embodiment, the Wide Multiply Matrix instructions use a wide operand to specify a matrix of values of width up to 64 bits (one half of register file and data path width) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 128 bits of symbols of twice the size of the source operand symbols. The width and depth of the matrix can be selected by specifying the size and shape of the wide operand as described above. Controls within the instruction allow specification of signed, ~~mixed-signed,~~mixed signed, unsigned, complex, or polynomial operands.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1410 of the Wide Multiply Matrix instruction is shown in Fig. 14A.

An exemplary embodiment of the schematics 1430 and 1460 of the Wide Multiply Matrix instruction is shown in Figs. 14B and 14C. In an exemplary embodiment, the

contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is catenated and placed in register rd.

In an exemplary embodiment, the memory multiply instructions (W.MUL.MAT, W.MUL.MAT.C, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) perform a partitioned array multiply of up to 8192 bits, that is 64x128 bits. The width of the array can be limited to 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired size in bytes to the virtual address operand: 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 1024/gsize bytes (or 512/gsize for W.MUL.MAT.C) (with gsize measured in bits), or must be the sum of an aligned address and one half of the size of the memory operand in bytes and/or one quarter of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

In an exemplary embodiment, a wide multiply octlets instruction (W.MUL.MAT.type.64, type=NONE M U P) is not implemented and causes a reserved instruction exception, as an ensemble-multiply-sum-octlets instruction (E.MUL.SUM.type.64) performs the same operation except that the multiplier is sourced from a 128-bit register rather than memory. Similarly, instead of wide-multiply-complex-quadlets instruction (W.MUL.MAT.C.32), one should use an ensemble-multiply-complex-quadlets instruction (E.MUL.SUM.C.32).

As shown in Fig. 14B, an exemplary embodiment of a wide-multiply-doublets instruction (W.MUL.MAT, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) multiplies memory [m31 m30 ... ml m0] with vector [h g f e d c b a], yielding products [hm31+gm27+...+bm7+am3 ... hm28+gm24+...+bm4+am0].

As shown in Fig. 14C, an exemplary embodiment of a wide-multiply-matrix-complex-doublets instruction (W.MUL.MAT.C) multiplies memory [ml5 m14 ... ml m0]

with vector [h g f e d c b a], yielding products [hm14+gm15+...+bm2+am3 ... hml2+gml3+...+bm0+aml hml3+gml2+... bml+am0].

An exemplary embodiment of the pseudocode 1480 of the Wide Multiply Matrix instruction is shown in Fig. 14D. An exemplary embodiment of the exceptions 1490 of the Wide Multiply Matrix instruction is shown in Fig. 14E.

Wide Multiply Matrix Extract

TheAn exemplary embodiment of the Wide Multiply Matrix Extract instruction is shown in Figures 15A-15F. In an exemplary embodiment, the Wide Multiply Matrix Extract instructions use a wide operand to specify a matrix of value of width up to 128 bits (full width of register file and data path) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 256 bits of symbols of twice the size of the source operand symbols plus additional bits to represent the sums of products without overflow. The results are then extracted in a manner described below (Enhanced Multiply Bandwidth by Result Extraction), as controlled by the contents of a general register specified by the instruction. The general register also specifies the format of the operands: signed, mixed-signed, unsigned, and complex as well as the size of the operands, byte (8 bit), doublet (16 bit), quadlet (32 bit), or hexlet (64 bit).

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1510 of the Wide Multiply Matrix Extract instruction is shown in Fig. 15A.

An exemplary embodiment of the schematics 1530 and 1560 of the Wide Multiply Matrix Extract instruction is shown in Figs. 15C and 14D. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rd. The group size and other parameters are specified from the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed, producing a

group of values. The group of values is rounded, and limited as specified, yielding a group of results which is the size specified. The group of results is catenated and placed in register ra.

In an exemplary embodiment, the size of this operation is determined from the contents of register rb. The multiplier usage is constant, but the memory operand size is inversely related to the group size. Presumably this can be checked for cache validity.

In an exemplary embodiment, low order bits of re are used to designate a size, which must be consistent with the group size. Because the memory operand is cached, the size can also be cached, thus eliminating the time required to decode the size, whether from rb or from rc.

In an exemplary embodiment, the wide multiply matrix extract instructions (W.MUL.MAT.X.B, W.MUL.MAT.X.L) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired memory operand size in bytes to the virtual address operand.

As shown in Fig. 15B, in an exemplary embodiment, bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted. The position and default values of the control fields allow for the source position to be added to a fixed control value for dynamic computation, and allow for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI instruction.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	mixed-sign vs. same-sign multiplication
1	1	saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9 bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512 4***gsize+spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**) 1.

In an exemplary embodiment, the values in the **s, n, m, t,** and **rnd** fields have the following meaning:

values	s	n	m	I	rnd
0	unsigned	real	same-sign	truncate	F
1	signed	complex	mixed-sign	saturate	Z
2					N
3					C

In an exemplary embodiment, the virtual address must be aligned, that is, it must be an exact multiple of the operand size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

In an exemplary embodiment, Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

As shown in Fig. 15C, an exemplary embodiment of a wide-multiply-matrix-extract-doublets instruction (W.MUL.MAT.X.B or W.MUL.MAT.X.L) multiplies

memory [m63 m62 m61 ... m2 ml m0] with vector [h g f e d c b a], yielding the products

[am7+bm15+cm23+dm31 +em39+fm47+gm55+hm63 . . .

am2+bm10+cm18+dm26+em34+fm42+gm50+hm58

aml+bm9+cm17+dm25+em33+fm41+gm49+hm57

am0+bm8+cml6+dm24+em32+fm40+gm48+hm56], rounded and limited as specified.

As shown in Fig. 15D, an exemplary embodiment of a wide-multiply-matrix-extract-complex-doublets instruction (W.MUL.MAT.X with n set in rb) multiplies memory [m31 m30 m29 ... m2 ml m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 ... am2-bm3+cm10-dm11+eml8-fm19+gm26-hm27 am1+bm0+cm9+dm8+em17+fm16+gm25+hm24 am0-bm1+cm8-dm9+em16-f17+gm24 hm25], rounded and limited as specified.

An exemplary embodiment of the pseudocode 1580 of the Wide Multiply Matrix Extract instruction is shown in Fig. 15E. An exemplary embodiment of the exceptions 1590 of the Wide Multiply Matrix Extract instruction is shown in Fig. 15F.

Wide Multiply Matrix Extract Immediate

An exemplary embodiment of the Wide Multiply Matrix Extract Immediate instruction is shown in Figures 16A-16E. In an exemplary embodiment, the Wide Multiply Matrix Extract Immediate instructions perform the same function as above, except that the extraction, operand format and size is controlled by fields in the instruction. This form encodes common forms of the above instruction without the need to initialize a register with the required control information. Controls within the instruction allow specification of signed, ~~mixed-signed,~~mixed signed, unsigned, and complex operands.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1610 of the Wide Multiply Matrix Extract Immediate instruction is shown in Fig. 16A.

An exemplary embodiment of the schematics 1630 and 1660 of the Wide Multiply Matrix Extract Immediate instruction is shown in Figs. 16B and 16C. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of

specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed in columns, producing a group of sums. The group of sums is rounded, limited, and extracted as specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd. All results are signed, N (nearest) rounding is used, and all results are limited to maximum representable signed values.

In an exemplary embodiment, the wide-multiply-extract-immediate-matrix instructions (W.MUL.MAT.X.I, W.MUL.MAT.X.I.C) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.X.I.C), or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in Fig. 16B, an exemplary embodiment of a wide-multiply-extract-immediate-matrix-doublets instruction (W.MUL.MAT.X.I.16) multiplies memory [m63 m62 m61 ... m2 m1 m0] with vector [h g f e d c b a], yielding the products

[am7+bm 15+cm23+dm31+em39+fm47+gm55+hm63 . . .
am2+bm10+cm 18+dm26+em34+fm42+gm50+hm58
am1+bm9+cm17+dm25+em33+fm41+gm49+hm57
am0+bm8+cm16+dm24+em32+fm40+gm48+hm56], rounded and limited as specified.

As shown in Fig. 16C, an exemplary embodiment of a wide-multiply-matrix-extract-immediate-complex-doublets instruction (W.MUL.MAT.X.I.C.16) multiplies memory [m31 m30 m29 ... m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 ... am2-bm3+cm10-dm11+em18-

fm19+gm26-hm27 am1+bm0+cm9+dm8+em17+fml6+gm25+hm24 am0-bm1+cm8-dm9+em16-f17+gm24-hm25], rounded and limited as specified.

An exemplary embodiment of the pseudocode 1680 of the Wide Multiply Matrix Extract Immediate instruction is shown in Fig. 16D. An exemplary embodiment of the exceptions 1590 of the Wide Multiply Matrix Extract Immediate instruction is shown in Fig. 16E.

Wide Multiply Matrix Floating-point

An exemplary embodiment of the Wide Multiply Matrix Floating-point instruction is shown in Figures 17A-17E. In an exemplary embodiment, the Wide Multiply Matrix Floating-point instructions perform a matrix multiply in the same form as above, except that the multiplies and additions are performed in floating-~~floating~~-point arithmetic. Sizes of half (16-bit), single (32-bit), double (64-bit), and complex sizes of half, single and double can be specified within the instruction.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1710 of the Wide Multiply Matrix Floating point instruction is shown in Fig. 17A.

An exemplary embodiment of the schematics 1730 and 1760 of the Wide Multiply Matrix Floating-point instruction is shown in Figs. 17B and 17C. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is catenated and placed in register rd.

In an exemplary embodiment, the wide-multiply-matrix-floating-point instructions (W.MUL.MAT.F, W.MUL.MAT.C.F) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual

address operand: 8, 4, or 2. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.C.F), or must be the sum of an aligned address and one half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in Fig. 17B, an exemplary embodiment of a wide-multiply-matrix-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m31 m30 ... m1 m0] with vector [h g f e d c b a], yielding products [hm31+gm27+...+bm7+am3 ... hm28+gm24+. . .+bm4+am0].

As shown in Fig. 17C, an exemplary embodiment of a wide-multiply-matrix-complex-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m15 m14 ... m1 m0] with vector [h g f e d c b a], yielding products [hm14+gm15+...+bm2+am3 ... hm12+gm13+...+bm0+am1 -hm13+gm12+...-bm1+am0].

An exemplary embodiment of the pseudocode 1780 of the Wide Multiply Matrix Floating-point instruction is shown in Fig. 17D. Additional pseudocode functions used by this and other floating point instructions is shown in Figure FLOAT-1. An exemplary embodiment of the exceptions 1790 of the Wide Multiply Matrix Floating-point instruction is shown in Fig. 17E.

Wide Multiply Matrix Galois

An exemplary embodiment of the Wide Multiply Matrix Galois instruction is shown in Figures 18A-18D. In an exemplary embodiment, the Wide Multiply Matrix Galois instructions perform a matrix multiply in the same form as above, except that the multiples and additions are performed in Galois field arithmetic. A size of 8 bits can be specified within the instruction. The contents of a general register specify the polynomial with which to perform the Galois field remainder operation. The nature of the matrix multiplication is novel and described in detail below.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, second and third operands from general registers, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1810 of the Wide Multiply Matrix Galois instruction is shown in Fig. 18A.

An exemplary embodiment of the schematic 1830 of the Wide Multiply Matrix Galois instruction is shown in Fig. 18B. In an exemplary embodiment, the contents of register re is used as a virtual address, and a value of specified size is loaded from memory. Second and third values are the contents of registers rd and rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied as polynomials with the first value, producing a result which is reduced to the Galois field specified by the third value, producing a group of result values. The group of result values is catenated and placed in register ra.

In an exemplary embodiment, the wide-multiply-matrix-Galois-bytes instruction (W.MUL.MAT.G.8) performs a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 256 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in Fig. 18B, an exemplary embodiment of a wide-multiply-matrix-Galois-byte instruction (W.MUL.MAT.G.8) multiplies memory [m255 m254 ... m1 m0] with vector [p o n m l k j i h g f e d c b a], reducing the result modulo polynomial [q], yielding products [(pm255+om247+...+bm31+am15 mod q) (pm254+om246+...+bm30+am14 mod q) ... (pm248+om240+...+bm16+am0 mod q)].

An exemplary embodiment of the pseudocode 1860 of the Wide Multiply Matrix

Galois instruction is shown in Fig. 18C. An exemplary embodiment of the exceptions 1890 of the Wide Multiply Matrix Galois instruction is shown in Fig. 18D.

Memory Operands of Either Little-Endian or Big-Endian Conventional Byte Ordering

In another aspect of the invention, memory operands of either little-endian or big-endian conventional byte ordering are facilitated. Consequently, all Wide operand instructions are specified in two forms, one for little-endian byte ordering and one for big-endian byte ordering, as specified by a portion of the instruction. The byte order specifies to the memory system the order in which to deliver the bytes within units of the data path width (128 bits), as well as the order to place multiple memory words (128 bits) within a larger Wide operand. Each of these instructions is described in greater detail in the System Architecture/BroadMX operand.

Architecture descriptions included herein.

Extraction of a High Order Portion of a Multiplier Product or Sum of Products

Another aspect of the present invention addresses extraction of a high order portion of a multiplier product or sum of products, as a way of efficiently utilizing a large multiplier array. Related U.S. PatentNo. 5,742,840 and U.S. Patent Application No. 08/857596 (notice of allowance 11/13/98),5,953,241 describe a system and method for enhancing the utilization of a multiplier array by adding specific classes of instructions to a general-purpose processor. This addresses the problem of making the most use of a large multiplier array that is fully used for high-precision arithmetic – for example a 64x64 bit multiplier is fully used by a 64-bit by 64-bit multiply, but only one quarter used for a 32-bit by 32-bit multiply) for (relative to the multiplier data width and registers) low-precision arithmetic operations. In particular, operations that perform a great many low-precision multiplies which are combined (added) together in various ways are specified. One of the overriding considerations in selecting the set of operations is a limitation on the size of the result operand. In an exemplary embodiment, for example, this size might be limited to on the order of 128 bits, or a single register, although no specific size limitation need exist.

The size of a multiply result, a product, is generally the sum of the sizes of the

operands, multiplicands and multiplier. Consequently, multiply instructions specify operations in which the size of the result is twice the size of identically-sized input operands. For our prior art design, for example, a multiply instruction accepted two 64-bit register sources and produces a single 128-bit register-pair result, using an entire 64x64 multiplier array for 64-bit symbols, or half the multiplier array for pairs of 32-bit symbols, or ~~one-quarter~~one quarter the multiplier array for quads of 16-bit symbols. For all of these cases, note that two register sources of 64 bits are combined, yielding a 128-bit result.

In several of the operations, including complex multiplies, convolve, and matrix multiplication, low-precision multiplier products are added together. The additions further increase the required precision. The sum of two products requires one additional bit of precision; adding four products requires two, adding eight products requires three, adding sixteen products requires four. In some prior designs, some of this precision is lost, requiring scaling of the multiplier operands to avoid overflow, further reducing accuracy of the result.

The use of register pairs creates an undesirable complexity, in that both the register pair and individual register values must be bypassed to subsequent instructions. As a result, with prior art techniques only half of the source operand 128-bit register values could be employed toward producing a single-register 128-bit result.

In the present invention, a high-order portion of the multiplier product or sum of products is extracted, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion as round-to-nearest/even, toward zero, floor, or ceiling. Overflows are handled by limiting the result to the largest and smallest values that can be accurately represented in the output ~~result. This operation is more fully described in the System Architecture/BroadMX Architecture descriptions included herein.~~result .

Extract Controlled by a Register

In the present invention, when the extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled to be used in subsequent operations without concern of overflow or rounding, enhancing performance.

Also in the present invention, when the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing all this control information in a single register, the size of the instruction is reduced over the number of bits that such a instruction would otherwise require, improving performance and enhancing flexibility of the processor.

The particular instructions included in this aspect of the present invention are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract and Ensemble Scale Add ~~Extract. each of which is more thoroughly treated in the System Architecture/BroadMX Architecture descriptions included~~ Extract.

~~herein..~~

~~An aspect of the present invention defines the Ensemble Scale Add Extract instruction, that combines the extract control information in a register along with two values that are used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers that would otherwise be required, or the number of bits that the instruction would otherwise require, improving performance.~~

Ensemble Extract Inplace

~~Several~~An exemplary embodiment of the Ensemble Extract Inplace instruction is shown in Figures 19A-19G. In an exemplary embodiment, several of these instructions (Ensemble Convolve Extract, Ensemble Multiply Add Extract) are typically available only in forms where the extract is specified as part of the instruction. An **alternative** embodiment can incorporate forms of the operations in which the size of the operand, the shift amount and the rounding can be controlled by the contents of a general register (as they are in the Ensemble Multiply Extract instruction). The definition of this kind of instruction for Ensemble Convolve Extract, and Ensemble Multiply Add Extract would require four source registers, which increases complexity by requiring additional general-register read ports.

In an exemplary embodiment, these operations take operands from four registers, perform operations on partitions of bits in the operands, and place the concatenated results in a fourth register. An exemplary embodiment of the format and operation codes 1910 of the Ensemble Extract Inplace instruction is shown in Fig. 19A.

An exemplary embodiment of the schematics 1930, 1945, 1960, and 1975 of the Ensemble Extract Inplace instruction is shown in Figs. 19C, 19D, 19E, and 19F. In an exemplary embodiment, the contents of registers rd, rc, rb, and ra are fetched. The specified operation is performed on these operands. The result is placed into register rd.

In an exemplary embodiment, for the E.CON.X instruction, the contents of registers rd and rc are catenated, as c d, and used as a first value. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are convolved, producing a group of values. The group of values is rounded, limited and extracted as specified, yielding a group of results that is the size specified. The group of results is catenated and placed in register rd.

In an exemplary embodiment, for the E.MUL.ADD.X instruction, the contents of registers rc and rb are partitioned into groups of operands of the size specified and are multiplied, producing a group of values to which are added the partitioned and extended contents of register rd. The group of values is rounded, limited and extracted as specified, yielding a group of results that is the size specified. The group of results is catenated and placed in register rd.

As shown in Fig. 19B, in an exemplary embodiment, bits 31..0 of the contents of register **ra** specifies several parameters that control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction. The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	extended vs. group size result
s	1	signed vs. unsigned
n	1	complex vs. real multiplication

m	1	mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize+spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

In an exemplary embodiment, the values in the **x, s, n, m, l**, and **rnd** fields have the following meaning:

values	x	s	n	m	l	rnd
0	group	unsigned	real	same-sign	truncate	F
1	extended	signed	complex	mixed-sign	saturate	Z
2						N
3						C

Ensemble Multiply Add Extract

As shown in Fig. 19C, an exemplary embodiment of an ensemble-multiply-add-extract-doublets instruction (E.MULADDX) multiplies vector rc [h g f e d c b a] with vector rb [p o n m l k j i], and adding vector rd [x w v u t s r q], yielding the result vector rd [hp+x go+w fn+v em+u dl+t ck+s bj+r ai+q], rounded and limited as specified by ra31..0.

As shown in Fig. 19D, an exemplary embodiment of an ensemble-multiply-add-extract-doublets-complex instruction (E.MUL.X with n set) multiplies operand vector rc [h g f e d c b a] by operand vector rb [p o n m l k j i], yielding the result vector rd [gp+ho go-hp en+fm em-fn cl+dk ck-dl aj+bi ai-bj], rounded and limited as specified by ra31..0. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.

Ensemble Convolve Extract

As shown in Fig. 19E, an exemplary embodiment of an ensemble-convolve-

extract-doublets instruction (ECON.X with n=0) convolves vector rc r d [x w v u t s r q p o n m l k j i] with vector rb [h g f e d c b a], yielding the products vector rd

[ax+bw+cv+du+et+fs+gr+hq ... as+br+cq+dp+eo+fn+gm+hl

ar+bq+cp+do+en+fm+gl+hk aq+bp+co+dn+em+fl+gk+hj], rounded and limited as specified by $ra_{31..0}$.

As shown in Fig. 19F, an exemplary embodiment of an ensemble-convolve-extract-complex-doublets instruction (ECON.X with n=1) convolves vector rd r c [x w v u t s r q p o n m l k j i] with vector rb [h g f e d c b a], yielding the products vector rd

[ax+bw+cv+du+et+fs+gr+hq ... as-bt+cq-dr+eo-fp+gm-hn

ar+bq+cp+do+en+fm+gl+hk aq-br+co-dp+em-fn+gk+hl], rounded and limited as specified by ra31..0.

An exemplary embodiment of the pseudocode 1990 of Ensemble Extract Inplace instruction is shown in Fig. 19G. In an exemplary embodiment, there are no exceptions for the Ensemble Extract Inplace instruction.

Ensemble Extract

An exemplary embodiment of the Ensemble Extract instruction is shown in Figures 20A-20J. In an exemplary embodiment, these operations take operands from three registers, perform operations on partitions of bits in the operands, and place the catenated results in a fourth register. An exemplary embodiment of the format and operation codes 2010 of the Ensemble Extract instruction is shown in Fig. 20A.

An exemplary embodiment of the schematics 2020, 2030, 2040, 2050, 2060, 2070, and 2080 of the Ensemble Extract Inplace instruction is shown in Figs. 20C, 20D, 20E, 20F, 20G, 20H, and 20I. In an exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

As shown in Fig. 20B, in an exemplary embodiment, bits 31..0 of the contents of register rb specifies several parameters that control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for

some of the simpler extract cases by a single GCOPYI.128 instruction. The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	extended vs. group size result
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	merge vs. extract or mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512 4***gsize+spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

In an exemplary embodiment, the values in the **x, s, n, m, l**, and **rnd** fields have the following meaning:

values	x	s	n	m	l	rnd
0	group	unsigned	real	extract/same-sign	truncate	F
1	extended	signed	complex	merge/mixed-sign	saturate	Z
2						N
3						C

In an exemplary embodiment, for the E.SCAL.ADD.X instruction, bits 127..64 of the contents of register rb specifies the multipliers for the multiplicands in registers rd and rc. Specifically, bits 64+2*gsize-1..64+gsize is the multiplier for the contents of register rd, and bits 64+gsize-1..64 is the multiplier for the contents of register rc.

Ensemble Multiply Extract

As shown in Fig. 20C, an exemplary embodiment of an ensemble-multiply-extract-doublets instruction (E.MULX) multiplies vector rd [h g f e d c b a] with vector rc [p o n m l k j i], yielding the result vector ra [hp go fn em dl ck bj ai], rounded and limited as specified by $rb_{31..0}$.

As shown in Fig. 20D, an exemplary embodiment of an ensemble-multiply-extract-doublets-complex instruction (E.MUL.X with n set) multiplies vector rd [h g f e d c b a] by vector rc [p o n m l k j i], yielding the result vector ra [gp+ho go-hp en+fm em-fn cl+dk ck-dl aj+bi ai-bj], rounded and limited as specified by $rb_{31..0}$. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.

Ensemble Scale Add Extract

An aspect of the present invention defines the Ensemble Scale Add Extract instruction, that combines the extract control information in a register along with two values that are used as scalar multipliers to the contents of two vector multiplicands.

This combination reduces the number of registers that would otherwise be required, or the number of bits that the instruction would otherwise require, improving performance. Another advantage of the present invention is that the combined operation may be performed by an exemplary embodiment with sufficient internal precision on the summation node that no intermediate rounding or overflow occurs, improving the accuracy over prior art operation in which more than one instruction is required to perform this computation.

As shown in Fig. 20E, an exemplary embodiment of an ensemble-scale-add-extract-doublets instruction (E.SCAL.ADD.X) multiplies vector rd [h g f e d c b a] with $rb_{95..80}$ [r] and adds the product to the product of vector rc [p o n m l k j i] with $rb_{79..64}$ [q], yielding the result [hr+pq gr+oq fr+nq er+mq dr+lq cr+kq br+jq ar+iq], rounded and limited as specified by $rb_{31..0}$.

As shown in Fig. 20F, an exemplary embodiment of an ensemble-scale-add-extract-doublets-complex instruction (E.SCLADD.X with n set) multiplies vector rd [h g f e d c b a] with $rb_{127..96}$ [t s] and adds the product to the product of vector rc [p o n m l k j i] with $rb_{95..64}$ [r q], yielding the result [hs+gt+pq+or gs-ht+oq-pr fs+et+nq+mr es-ft+mq-nr ds+ct+lq+kr cs-dt+kq-lr bs+at+jq+ir as-bt+iq-jr], rounded and limited as specified by $rb_{31..0}$.

Ensemble Extract

As shown in Fig. 20G, in an exemplary embodiment, for the E.EXTRACT instruction, when m=0 and x=0, the parameters specified by the contents of register rb are interpreted to select fields from double size symbols of the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra.

As shown in Fig. 20H, in an exemplary embodiment, for an ensemble-merge-extract (E.EXTRACT when m=1), the parameters specified by the contents of register rb are interpreted to merge fields from symbols of the contents of register rd with the contents of register rc. The results are catenated and placed in register ra. The x field has no effect when m=1.

As shown in Fig. 20I, in an exemplary embodiment, for an ensemble-expand-extract (E.EXTRACT when m=0 and x=1), the parameters specified by the contents of register rb are interpreted to extract fields from symbols of the contents of register rd. The results are catenated and placed in register ra. Note that the value of rc is not used.

An exemplary embodiment of the pseudocode 2090 of Ensemble Extract instruction is shown in Fig. 20J. In an exemplary embodiment, there are no exceptions for the Ensemble Extract instruction.

Reduction of Register Read Ports

Another alternative embodiment can reduce the number of register read ports required for implementation of instructions in which the size, shift and rounding of operands is controlled by a register. The value of the extract control register can be fetched using an additional cycle on an initial execution and retained within or near the functional unit for subsequent executions, thus reducing the amount of hardware required for implementation with a small additional performance penalty. The value retained

would be marked invalid, causing a re-fetch of the extract control register, by instructions that modify the register, or alternatively, the retained value can be updated by such an operation. A re-fetch of the extract control register would also be required if a different register number were specified on a subsequent execution. It should be clear that the properties of the above two alternative embodiments can be combined.

Galois Field Arithmetic

Another aspect of the invention includes Galois field arithmetic, where multiplies are performed by an initial binary polynomial multiplication (unsigned binary multiplication with carries suppressed), followed by a polynomial modulo/remainder operation (unsigned binary division with carries suppressed). The remainder operation is relatively expensive in area and delay. In Galois field arithmetic, additions are performed by binary addition with carries suppressed, or equivalently, a bitwise ~~exclusive-or~~exclusive or operation. In this aspect of the present invention, a matrix multiplication is performed using Galois field arithmetic, where the multiplies and additions are Galois field multiples and additions.

Using prior art methods, a 16 byte vector ~~multipled by a 16x16~~multiplied by a 16x 16 byte matrix can be performed as 256 8-bit Galois field multiplies and 16*_15=240 8-bit Galois field additions. Included in the 256 Galois field multiplies are 256 polynomial multiplies and 256 polynomial remainder operations.

~~operations. But by~~By use of the present invention, the total computation ~~can be~~is reduced significantly by performing 256 polynomial multiplies, 240 16-bit polynomial additions, and 16 polynomial remainder operations. Note that the cost of the polynomial additions has been doubled compared with the Galois field additions, as these are now 16-bit operations rather than 8-bit operations, but the cost of the polynomial remainder functions has been reduced by a factor of 16. Overall, this is a favorable tradeoff, as the cost of addition is much lower than the cost of remainder.

~~In a still further aspect of the present invention, a technique is provided for incorporating floating point information into processor instructions. In related US patent 5812439, a system and method are described for incorporating control of rounding and exceptions for floating point instructions into the instruction itself. The present invention extends this invention to include separate instructions in which rounding is specified, but~~

~~default handling of exceptions is also specified, for a particular class of floating-point instructions. Specifically, the SINK instruction (which converts floating-point values to integral values) is available with control in the instruction that include all previously specified combinations (default-near rounding and default exceptions, Z – round toward-zero and trap on exceptions, N – round to nearest and trap on exceptions, F – floor rounding (toward minus infinity) and trap on exceptions, C – ceiling rounding (toward plus infinity) and trap on exceptions, and X – trap on inexact and other exceptions), as well as three new combinations (Z.D – round toward zero and default exception handling, F.D – floor rounding and default exception handling, and C.D – ceiling rounding and default exception handling). (The other combinations: N.D is equivalent to the default, and X.D – trap on inexact but default handling for other exceptions is possible but not particularly valuable).~~**Decoupled Access from Execution Pipelines and Simultaneous Multithreading**

In yet another aspect of the present invention, best shown in Figure 4, the present invention employs both decoupled access from execution pipelines and simultaneous multithreading in a unique way. Simultaneous Multithreaded pipelines have been employed in prior art to enhance the utilization of data path units by allowing instructions to be issued from one of several execution threads to each functional ~~unit. (e.g., Susan Eggers, University of Wash, papers on Simultaneous Multithreading).~~

unit (e.g. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On Chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June, 1995).

Decoupled access from execution pipelines have been employed in prior art to enhance the utilization of execution data path units by buffering results from an access unit, which computes addresses to a memory unit that in turn fetches the requested items from memory, and then presenting them to an execution unit (e.g., ~~James E. Smith, paper on Decoupled Access from Execution).~~J. E. Smith, "Decoupled Access/Execute Computer Architectures", Proceedings of the Ninth Annual International Symposium on Computer Architecture, Austin, Texas (April 26 29, 1982), pp. 112-119).

Compared to conventional pipelines, the Eggers prior art used an additional pipeline cycle before instructions could be issued to functional units, the additional cycle

needed to determine which threads should be permitted to issue instructions. Consequently, relative to conventional pipelines, the prior art design had additional delay, including dependent branch delay.

The present invention contains individual access data path units, with associated register files, for each execution thread. These access units produce addresses, which are aggregated together to a common memory unit, which fetches all the addresses and places the memory contents in one or more buffers. Instructions for execution units, which are shared to varying degrees among the threads are also buffered for later execution. The execution units then perform operations from all active threads using functional data path units that are shared.

For instructions performed by the execution units, the extra cycle required for prior art simultaneous multithreading designs is overlapped with the memory data access time from prior art decoupled access from execution cycles, so that no additional delay is incurred by the execution functional units for scheduling resources. For instructions performed by the access units, by employing individual access units for each thread the additional cycle for scheduling shared resources is also eliminated.

This is a favorable tradeoff because, while threads do not share the access functional units, these units are relatively small compared to the execution functional units, which are shared by threads.

With regard to the sharing of execution units, the present invention employs several different classes of functional units for the execution unit, with varying cost, utilization, and performance. In particular, the G units, which perform simple addition and bitwise operations is relatively inexpensive (in area and power) compared to the other units, and its utilization is relatively high. Consequently, the design employs four such units, where each unit can be shared between two threads. The X unit, which performs a broad class of data switching functions is more expensive and less used, so two units are provided that are each shared among two threads. The T unit, which performs the Wide Translate instruction, is expensive and utilization is low, so the single unit is shared among all four threads. The E unit, which performs the class of Ensemble instructions, is very expensive in area and power compared to the other functional units, but utilization is relatively high, so we provide two such units, each unit shared by two threads.

In Figure 4, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 401-404, coupled to an access register file AR 405-408, each of which is, in turn, coupled to two access functional units A 409-416. The access units function independently for four simultaneous threads of execution. These eight access functional units A 409-416 produce results for access register files AR 405-408 and addresses to a shared memory system 417. The memory contents fetched from memory system 417 are combined with execute instructions not performed by the access unit and entered into the four execute instruction queues E-Queue 421-424. Instructions and memory data from E-queue 421-424 are presented to execution register files 425-428, which fetches execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 431, that selects which instructions from the four threads are to be routed to the available execution units E 441 and 449, X 442 and 448, G 443-444 and 446-447, and T 445. The execution register file source operands ER 425-428 are coupled to the execution units 441-445 using source operand buses 451-454 and to the execution units 445-449 using source operand buses 455-458. The function unit result operands from execution units 441-445 are coupled to the execution register file using result bus 461 and the function units result operands from execution units 445-449 are coupled to the execution register file using result bus 462.

~~The foregoing elements of the present invention may be better understood with reference to the System Architecture/BroadMX Architecture descriptions included herein.~~

Improved Interprivilege Gateway

In a still further aspect of the present invention, an improved interprivilege gateway is described which involves increased parallelism and leads to enhanced performance. In related U.S. Patent Application No. 08/541,416, a system and method is described for implementing an instruction that, in a controlled fashion, allows the transfer of control (branch) from a ~~lower privilege level to a higher privilege~~lower privilege level to a higher privilege level. The present invention is an improved system and method for a modified instruction that accomplishes the same purpose but with specific advantages.

Many processor resources, such as control of the virtual memory system itself, input and output operations, and system control functions are protected from accidental or

malicious misuse by enclosing them in a protective, privileged region. Entry to this region must be established only though particular entry points, called gateways, to maintain the integrity of these protected regions.

Prior art versions of this operation generally load an address from a region of memory using a protected virtual memory attribute that is only set for data regions that contain valid gateway entry points, then perform a branch to an address contained in the contents of memory. Basically, three steps were involved: load, ~~branch, then~~then branch and check. Compared to other instructions, such as ~~register-to-register~~register to register computation instructions and memory loads and stores, and ~~register-based~~register based branches, this is a substantially longer operation, which introduces delays and complexity to a pipelined implementation.

In the present invention, the branch-gateway instruction performs two operations in parallel: 1) a branch is performed to the ~~contents~~Contents of register 0 and 2) a load is performed using the contents of register 1, using a specified byte order (little-endian) and a specified size (64 bits). If the value loaded from memory does not equal the contents of register 0, the instruction is aborted due to an exception. In addition, 3) a return address (the next sequential instruction address following the branch-gateway instruction) is written into register 0, provided the instruction is not aborted. This approach essentially uses a first instruction to establish the requisite permission to allow user code to access privileged code, and then a second instruction is permitted to branch directly to the privileged code because of the permissions issued for the first instruction.

In the present invention, the new privilege level is also contained in register 0, and the second parallel operation does not need to be performed if the new privilege level is not greater than the old privilege level. When this second operation is suppressed, the remainder of the instruction performs an identical function to a branch-link instruction, which is used for invoking procedures that do not require an increase in privilege. The advantage that this feature brings is that the branch-gateway instruction can be used to call a procedure that may or may not require an increase in privilege.

The memory load operation verifies with the virtual memory system that the region that is loaded has been tagged as containing valid gateway data. A further advantage of the present invention is that the called procedure may rely on the fact that register 1 contains the address that the gateway data was loaded from, and can use the contents of register 1 to locate additional data or addresses that the procedure may require. Prior art versions of this instruction required that an additional address be loaded from the gateway region of memory in order to initialize that address in a protected manner – the present invention allows the address itself to be loaded with a ~~"normal"~~"normal" load operation that does not require special protection.

The present invention allows a ~~"normal"~~"normal" load operation to also load the contents of register 0 prior to issuing the branch-gateway instruction. The value may be loaded from the same memory address that is loaded by the branch-gateway instruction, because the present invention contains a virtual memory system in which the region may be enabled for normal load operations as well as the special ~~"gateway"~~"gateway" load operation performed by the branch-gateway instruction.

Improved Interprivilege Gateway - System and Privileged Library Calls

An exemplary embodiment of the System and Privileged Library Calls is shown in Figures 21A-21 B. An exemplary embodiment of the schematic 2110 of System and Privileged Library Calls is shown in Fig. 21A. In an exemplary embodiment, it is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, a modified procedure call in which the process privilege level is quietly raised to the required level is used. To provide this mechanism safely, interaction with the virtual memory system is required.

In an exemplary embodiment, such a procedure must not be entered from anywhere other than its legitimate entry point, to prohibit entering a procedure after the point at which security checks are performed or with invalid register contents, otherwise the access to a higher privilege level can lead to a security violation. In addition, the procedure generally must have access to memory data, for which addresses must be produced by the privileged code. To facilitate generating these addresses, the branch-gateway instruction allows the privileged code procedure to rely on the fact that a single register has been verified to contain a pointer to a valid memory region.

In an exemplary embodiment, the branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that other registers such as the data pointer and stack pointer can be properly set. To ensure this, the branch-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway contains the virtual address of the entry point of the procedure and the target privilege level. A gateway can only exist in regions of the virtual address space designated to contain them, and can only be used to access privilege levels at or below the privilege level at which the memory region can be written to ensure that a gateway cannot be forged.

In an exemplary embodiment, the branch-gateway instruction ensures that register 1 (dp) contains a valid pointer to the gateway for this target code address by comparing the contents of register 0 (lp) against the gateway retrieved from memory and causing an exception trap if they do not match. By ensuring that register 1 points to the gateway, auxiliary information, such as the data pointer and stack pointer can be set by loading values located by the contents of register 1. For example, the eight bytes following the gateway may be used as a pointer to a data region for the procedure.

In an exemplary embodiment, before executing the branch-gateway instruction, register 1 must be set to point at the gateway, and register 0 must be set to the address of the target code address plus the desired privilege level. A "L.I.64.L.A r0=r1,0" instruction is one way to set register 0, if register 1 has already been set, but any means of getting the correct value into register 0 is permissible.

In an exemplary embodiment, similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities,

so a procedure may freely branch to a less-privileged code address. Normally, such a procedure restores the stack frame, then uses the branch-down instruction to return.

An exemplary embodiment of the typical dynamic-linked, inter-gateway calling sequence 2130 is shown in Fig. 21B. In an exemplary embodiment, the calling sequence is identical to that of the inter-module calling sequence shown above, except for the use of the B.GATE instruction instead of a B.LINK instruction. Indeed, if a B.GATE instruction is used when the privilege level in the lp register is not higher than the current privilege level, the B.GATE instruction performs an identical function to a B.LINK.

In an exemplary embodiment, the callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, as it can be forged. Similarly, any pointers which the callee provides should not be used directly unless it they are verified to point to regions which the callee should be permitted to address. This can be avoided by defining application programming interfaces (APIs) in which all values are passed and returned in registers, or by using a trusted, intermediate privilege wrapper routine to pass and return parameters. The method described below can also be used.

In an exemplary embodiment, it can be useful to have highly privileged code call less-privileged routines. For example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. To invoke the procedure, the privilege can be reduced via the branch-down instruction. The return from the procedure actually requires an increase in privilege, which must be carefully controlled. This is dealt with by placing the procedure call within a lower-privilege procedure wrapper, which uses the branch-gateway instruction to return to the higher privilege region after the call through a secure re-entry point. Special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame (or restoring the original lower-privilege stack) that may be manipulated by the less-privileged routine. Finally, such a technique is vulnerable to an unprivileged routine attempting to use the re-entry point directly, so it may be appropriate to keep a privileged state variable which controls permission to enter at the re-entry point.

Improved Interprivilege Gateway - Branch Gateway

An exemplary embodiment of the Branch Gateway instruction is shown in Figures 21C-21F. In an exemplary embodiment, this operation provides a secure means to call a procedure,

including those at a higher privilege level. An exemplary embodiment of the format and operation codes 2160 of the Branch Gateway instruction is shown in Fig. 21C.

An exemplary embodiment of the schematic 2170 of the Branch Gateway instruction is shown in Fig. 21D. In an exemplary embodiment, the contents of register rb is a branch address in the high-order 62 bits and a new privilege level in the low-order 2 bits. A branch and link occurs to the branch address, and the privilege level is raised to the new privilege level. The high-order 62 bits of the successor to the current program counter is catenated with the 2-bit current execution privilege and placed in register 0.

In an exemplary embodiment, if the new privilege level is greater than the current privilege level, an octlet of memory data is fetched from the address specified by register 1, using the little-endian byte order and a gateway access type. A GatewayDisallowed exception occurs if the original contents of register 0 do not equal the memory data.

In an exemplary embodiment, if the new privilege level is the same as the current privilege level, no checking of register 1 is performed.

In an exemplary embodiment, an AccessDisallowed exception occurs if the new privilege level is greater than the privilege level required to write the memory data, or if the old privilege level is lower than the privilege required to access the memory data as a gateway, or if the access is not aligned on an 8-byte boundary.

In an exemplary embodiment, a ReservedInstruction exception occurs if the rc field is not one or the rd field is not zero.

In an exemplary embodiment, in the example in Fig. 21 D, a gateway from level 0 to level 2 is illustrated. The gateway pointer, located by the contents of register rc (1), is fetched from memory and compared against the contents of register rb (0). The instruction may only complete if these values are equal. Concurrently, the contents of register rb (0) is placed in the program counter and privilege level, and the address of the next sequential address and privilege level is placed into register rd (0). Code at the target of the gateway locates the data pointer at an offset from the gateway pointer (register 1), and fetches it into register 1, making a data region available. A stack pointer may be saved and fetched using the data region, another region located from the data region, or a data region located as an offset from the original gateway pointer.

In an exemplary embodiment, this instruction gives the target procedure the assurances that register 0 contains a valid return address and privilege level, that register 1 points to the gateway location, and that the gateway location is octlet aligned. Register 1 can then be used to securely reach values in memory. If no sharing of literal pools is desired, register 1 may be used as a literal pool pointer directly. If sharing of literal pools is desired, register 1 may be used with an appropriate offset to load a new literal pool pointer; for example, with a one cache line offset from the register 1. Note that because the virtual memory system operates with cache line granularity, that several gateway locations must be created together.

In an exemplary embodiment, software must ensure that an attempt to use any octlet within the region designated by virtual memory as gateway either functions properly or causes a legitimate exception. For example, if the adjacent octlets contain pointers to literal pool locations, software should ensure that these literal pools are not executable, or that by virtue of being aligned addresses, cannot raise the execution privilege level. If register 1 is used directly as a literal pool location, software must ensure that the literal pool locations that are accessible as a gateway do not lead to a security violation.

In an exemplary embodiment, register 0 contains a valid return address and privilege level, the value is suitable for use directly in the Branch down (B.DOWN) instruction to return to the gateway callee.

An exemplary embodiment of the pseudocode 2190 of the Branch Gateway instruction is shown in Fig. 21E. An exemplary embodiment of the exceptions 2199 of the Branch Gateway instruction is shown in Fig. 21F.

Group Add

In accordance with one embodiment of the invention, the processor handles a variety fix-point, or integer, group operations. For example, Fig. 26A presents various examples of Group Add instructions accommodating different operand sizes, such as a byte (8 bits), doublet (16 bits), quadlet (32 bits), octlet (64 bits), and hexlet (128 bits). Figs. 26B and 26C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Add instructions shown in Fig. 26A. As shown in Figs. 26B and 26C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and added, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in

register rd. While the use of two operand registers and a different result register is described here and elsewhere in the present specification, other arrangements, such as the use of immediate values, may also be implemented.

In the present embodiment, for example, if the operand size specified is a byte (8 bits), and each register is 128-bit wide, then the content of each register may be partitioned into 16 individual operands, and 16 different individual add operations may take place as the result of a single Group Add instruction. Other instructions involving groups of operands may perform group operations in a similar fashion.

Group Set and Group Subtract

Similarly, Fig. 27A presents various examples of Group Set instructions and Group Subtract instructions accommodating different operand sizes. Figs. 27B and 27C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Set instructions and Group Subtract instructions. As shown in Figs. 27B and 27C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and for Group Set instructions are compared for a specified arithmetic condition or for Group Subtract instructions are subtracted, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

Ensemble Convolve, Divide, Multiply, Multiply Sum

In the present embodiment, other fix-point group operations are also available. Fig. 28A presents various examples of Ensemble Convolve, Ensemble Divide, Ensemble Multiply, and Ensemble Multiply Sum instructions accommodating different operand sizes. Figs. 28B and 28C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Ensemble Convolve, Ensemble Divide, Ensemble Multiply and Ensemble Multiply Sum instructions. As shown in Figs. 28B and 28C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and convolved or divided or multiplied, yielding a group of results, or multiplied and summed to a single result. The group of results is catenated and placed, or the single result is placed, in register rd.

Ensemble Floating-Point Add, Divide, Multiply, and Subtract

In accordance with one embodiment of the invention, the processor also handles a variety floating-point group operations accommodating different operand sizes. Here, the different operand sizes may represent floating point operands of different precisions, such as half-precision (16 bits), single-precision (32 bits), double-precision (64 bits), and quad-precision (128 bits). Fig. 29 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections and figures. In the functions set forth in Fig. 29, an internal format represents infinite-precision floating-point values as a four-element structure consisting of (1) s (sign bit): 0 for positive, 1 for negative, (2) t (type): NORM, ZERO, SNAN, QNAN, INFINITY, (3) e (exponent), and (4) f: (fraction). The mathematical interpretation of a normal value places the binary point at the units of the fraction, adjusted by the exponent: $(-1)^{s}*(2^{e})*f$. The function F converts a packed IEEE floating-point value into internal format. The function PackF converts an internal format back into IEEE floating-point format, with rounding and exception control.

Figs. 30A and 31A present various examples of Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. Figs. 30B-C and 31B-C illustrate an exemplary embodiment of formats and operation codes that can be used to perform the various Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. In these examples, Ensemble Floating Point Add, Divide, and Multiply instructions have been labeled as "EnsembleFloatingPoint." Also, Ensemble Floating-Point Subtract instructions have been labeled as "EnsembleReversedFloatingPoint." As shown in Figs. 30B-C and 31B-C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified, and the specified group operation is performed, yielding a group of results. The group of results is catenated and placed in register rd.

In the present embodiment, the operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Ensemble Scale-Add Floating-point

A novel instruction, Ensemble-Scale-Add improves processor performance by performing two sets of parallel multiplications and pairwise summing the products. This improves performance for operations in which two vectors must be scaled by two independent values and then summed, providing two advantages over nearest prior art operations of a fused-multiply-add. To perform this operation using prior art instructions, two instructions would be needed, an ensemble-multiply for one vector and one scaling value, and an ensemble-multiply-add for the second vector and second scaling value, and these operations are clearly dependent. In contrast, the present invention fuses both the two multiplies and the addition for each corresponding elements of the vectors into a single operation. The first advantage achieved is improved performance, as in an exemplary embodiment the combined operation performs a greater number of multiplies in a single operation, thus improving utilization of the partitioned multiplier unit. The second advantage achieved is improved accuracy, as an exemplary embodiment may compute the fused operation with sufficient intermediate precision so that no intermediate rounding the products is required.

An exemplary embodiment of the Ensemble Scale-Add Floating-point instruction is shown in Figures 22A-22B. In an exemplary embodiment, these operations take three values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register. An exemplary embodiment of the format 2210 of the Ensemble Scale-Add Floating-point instruction is shown in Fig. 22A.

In an exemplary embodiment, the contents of registers rd and rc are taken to represent a group of floating-point operands. Operands from register rd are multiplied with a floating-point operand taken from the least-significant bits of the contents of register rb and added to operands from register rc multiplied with a floating-point operand taken from the next least-significant bits of the contents of register rb. The results are rounded to the nearest representable floating-point value in a single floating-point operation. Floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. The results are catenated and placed in register ra.

An exemplary embodiment of the pseudocode 2230 of the Ensemble Scale-Add Floating-point instruction is shown in Fig. 22B. In an exemplary embodiment, there are no exceptions for the Ensemble Scale-Add Floating-point instruction.

Performing a Three-Input Bitwise Boolean Operation in a Single Instruction (Group Boolean)

In a further aspect of the present invention, a system and method is provided for performing a three-input bitwise Boolean operation in a single instruction. A novel method, ~~described in detail in the System Architecture/BroadMX Architecture descriptions herein,~~ is used to encode the eight possible output states of such an operation into only seven bits, and decoding these seven bits back into the eight states.

An exemplary embodiment of the Group Boolean instruction is shown in Figures 23A-23C. In an exemplary embodiment, these operations take operands from three registers, perform boolean operations on corresponding bits in the operands, and place the concatenated results in the third register. An exemplary embodiment of the format 2310 of the Group Boolean instruction is shown in Fig. 23A.

An exemplary embodiment of a procedure 2320 of Group Boolean instruction is shown in Fig. 23B. In an exemplary embodiment, three values are taken from the contents of registers rd, rc and rb. The ih and il fields specify a function of three bits, producing a single bit result. The specified function is evaluated for each bit position, and the results are catenated and placed in register rd. In an exemplary embodiment, register rd is both a source and destination of this instruction.

In an exemplary embodiment, the function is specified by eight bits, which give the result for each possible value of the three source bits in each bit position:

d	1 1 1 1 0 0 0 0
c	1 1 0 0 1 1 0 0
b	1 0 1 0 1 0 1 0
f(d,c,b)	f_7 f_6 f_5 f_4 f_3 f_2 f_1 f_0

In an exemplary embodiment, a function can be modified by rearranging the bits of the immediate value. The table below shows how rearrangement of immediate value $f_{7..0}$ can reorder the operands d,c,b for the same function.

operation	immediate
f(d,c,b)	f_7 f_6 f_5 f_4 f_3 f_2 f_1 f_0

f(c,d,b)	$f_7 f_6 f_3 f_2 f_5 f_4 f_1 f_0$
f(d,b,c)	$f_7 f_5 f_6 f_4 f_3 f_1 f_2 f_0$
f(b,c,d)	$f_7 f_3 f_5 f_1 f_6 f_2 f_4 f_0$
f(c,b,d)	$f_7 f_5 f_3 f_1 f_6 f_4 f_2 f_0$
f(b,d,c)	$f_7 f_3 f_6 f_2 f_5 f_1 f_4 f_0$

In an exemplary embodiment, by using such a rearrangement, an operation of the form: *b=f(d,c,b)* can be recoded into a legal form: *b=f(b,d,c)*. For example, the function: *b=f(d,c,b)=d?c:b* cannot be coded, but the equivalent function: *d=c?b:d* can be determined by rearranging the code for *d=f(d,c,b)=d?c: b*, which is 11001010, according to the rule for *f(d,c,b)* *f(c,b,d)*, to the code 11011000.

Encoding

In an exemplary embodiment, some special characteristics of this rearrangement is the basis of the manner in which the eight function specification bits are compressed to seven immediate bits in this instruction. As seen in the table above, in the general case, a rearrangement of operands from *f(d,c,b)* to *f(d,b,c)*.(interchanging rc and rb) requires interchanging the values of f_6 and f_5 and the values of f_2 and f_1.

In an exemplary embodiment, among the 256 possible functions which this instruction can perform, one quarter of them (64 functions) are unchanged by this rearrangement. These functions have the property that $f_6=f_5$ and $f_2=f_1$. The values of rc and rb (Note that rc and rb are the register specifiers, not the register contents) can be freely interchanged, and so are sorted into rising or falling order to indicate the value of f_2. (A special case arises when rc=rb, so the sorting of rc and rb cannot convey information. However, as only the values f_7, f_4, f_3, and f_0 can ever result in this case, f_6, f_5, f_2, and f_1 need not be coded for this case, so no special handling is required.) These functions are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field and f_2 by whether rc>rb, thus using 32 immediate values for 64 functions.

In an exemplary embodiment, another quarter of the functions have $f_6=1$ and $f_5=0$. These functions are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the quarter of the functions where $f_6=0$ and $f_5=1$, and are encoded by the values of f_7, f_4, f_3, f_2, f_1, and f_0 in the immediate field, thus using 64 immediate values for 128 functions.

In an exemplary embodiment, the remaining quarter of the functions have $f_6=f_5$ and f_2 f_1. The half of these in which $f_2=1$ and $f_1=0$ are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the eighth of the functions where $f_2=0$ and $f_1=1$, and are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field, thus using 32 immediate values for 64 functions.

In an exemplary embodiment, the function encoding is summarized by the table:

f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0	trc>trb	ih	il_5	il_4	il_3	il_2	il_1	il_0	rc	rb
		f_6				f_2		f_2	0	0	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6				f_2		$\sim f_2$	0	0	f_6	f_7	f_4	f_3	f_0	trb	trc
		f_6			0	1			0	1	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6			1	0			0	1	f_6	f_7	f_4	f_3	f_0	trb	trc
	0	1							1	f_2	f_1	f_7	f_4	f_3	f_0	trc	trb
	1	0							1	f_1	f_2	f_7	f_4	f_3	f_0	trb	trc

In an exemplary embodiment, the function decoding is summarized by the table:

ih	il_5	il_4	il_3	il_2	il_1	il_0	rc>rb	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
0	0						0	il_3	il_4	il_4	il_2	il_1	0	0	il_0
0	0						1	il_3	il_4	il_4	il_2	il_1	1	1	il_0
0	1							il_3	il_4	il_4	il_2	il_1	0	1	il_0
1								il_3	0	1	il_2	il_1	il_5	il_4	il_0

From the foregoing discussion, it can be appreciated that an exemplary embodiment of a compiler or assembler producing the encoded instruction performs the steps above to encode the instruction, comparing the f6 and f5 values and the f2 and fl values of the immediate field to determine which one of several means of encoding the immediate field is to be employed, and that the placement of the trb and trc register specifiers into the encoded instruction depends on the values of f2 (or f1) and f6 (or f5).

An exemplary embodiment of the pseudocode 2330 of the Group Boolean instruction is shown in Fig. 23C. It can be appreciated from the code that an exemplary embodiment of a circuit that decodes this instruction produces the f2 and fl values, when the immediate bits ih and

il5 are zero, by an arithmetic comparison of the register specifiers rc and rb, producing a one (1) value for f2 and fl when rc>rb. In an exemplary embodiment, there are no exceptions for the Group Boolean instruction.

Improving the Branch Prediction of Simple Repetitive Loops of Code

In yet a further aspect to the present invention, a system and method is described for improving the branch prediction of simple repetitive loops of code. In such a simple loop, the end of the loop is indicated by a conditional branch backward to the beginning of the loop. The condition branch of such a loop is taken for each iteration of the loop except the final iteration, when it is not taken. Prior art branch prediction systems have employed finite state machine operations to attempt to properly predict a majority of such conditional branches, but without specific information as to the number of times the loop iterates, will make an error in prediction when the loop terminates.

The system and method of the present invention includes providing a count field for indicating how many times a branch is likely to be taken before it is not taken, which enhances the ability to properly predict both the initial and final branches of simple loops when a compiler can determine the number of iterations that the loop will be performed. This improves performance by avoiding misprediction of the branch at the end of a loop when the loop terminates and instruction execution is to continue beyond the loop, as occurs in prior art branch prediction hardware.

Branch Hint

An exemplary embodiment of the Branch Hint instruction is shown in Figures 24A-24C. In an exemplary embodiment, this operation indicates a future branch location specified by a register.

In an exemplary embodiment, this instruction directs the instruction fetch unit of the processor that a branch is likely to occur **count** times at **simm** instructions following the current successor instruction to the address specified by the contents of register **rd**. An exemplary embodiment of the format 2410 of the Branch Hint instruction is shown in Fig. 24A.

In an exemplary embodiment, after branching count times, the instruction fetch unit presumes that the branch at **simm** instructions following the current successor instruction is not

likely to occur. If **count** is zero, this hint directs the instruction fetch unit that the branch is likely to occur more than 63 times.

In an exemplary embodiment, an Access disallowed exception occurs if the contents of register rd is not aligned on a quadlet boundary.

An exemplary embodiment of the pseudocode 2430 of the Branch Hint instruction is shown in Fig. 24B. An exemplary embodiment of the exceptions 2460 of the Branch Hint instruction is shown in Fig. 24C.

Incorporating Floating Point Information into Processor Instructions

In a still further aspect of the present invention, a technique is provided for incorporating floating point information into processor instructions. In related U.S. patent 581,2439, a system and method are described for incorporating control of rounding and exceptions for floating-point instructions into the instruction itself. The present invention extends this invention to include separate instructions in which rounding is specified, but default handling of exceptions is also specified, for a particular class of floating-point instructions.

Ensemble Sink Floating-point

In an exemplary embodiment, a Ensemble Sink Floating-point instruction, which converts floating-point values to integral values, is available with control in the instruction that include all previously specified combinations (default-near rounding and default exceptions, Z - round-toward-zero and trap on exceptions, N - round to nearest and trap on exceptions, F - floor rounding (toward minus infinity) and trap on exceptions, C - ceiling rounding (toward plus infinity) and trap on exceptions, and X - trap on inexact and other exceptions), as well as three new combinations (Z.D - round toward zero and default exception handling, F.D - floor rounding and default exception handling, and C.D - ceiling rounding and default exception handling). (The other combinations: N.D is equivalent to the default, and X.D - trap on inexact but default handling for other exceptions is possible but not particularly valuable).

An exemplary embodiment of the Ensemble Sink Floating-point instruction is shown in Figures 25A-25C. In an exemplary embodiment, these operations take one value from a register, perform a group of floating-point arithmetic conversions to integer on partitions of bits in the operands, and place the concatenated results in a register. An exemplary embodiment of the

operation codes, selection, and format 2510 of Ensemble Sink Floating-point instruction is shown in Fig. 25A.

In an exemplary embodiment, the contents of register rc is partitioned into floating-point operands of the precision specified and converted to integer values. The results are catenated and placed in register rd.

In an exemplary embodiment, the operation is rounded using the specified rounding option or using round-to- nearest if not specified. If a rounding option is specified, unless default exception handling is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified or if default exception handling is specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

An exemplary embodiment of the pseudocode 2530 of the Ensemble Sink Floating-point instruction is shown in Fig. 25B. An exemplary embodiment of the exceptions 2560 of the Ensemble Sink Floating-point instruction is shown in Fig. 25C.

An exemplary embodiment of the pseudocode 2570 of the Floating-point instructions is shown in Fig. 25D.

Crossbar Compress, Expand, Rotate, and Shift

In one embodiment of the invention, crossbar switch units such as units 142 and 148 perform data handling operations, as previously discussed. As shown in Fig. 32A, such data handling operations may include various examples of Crossbar Compress, Crossbar Expand, Crossbar Rotate, and Crossbar Shift operations. Figs. 32B and 32C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Crossbar Compress, Crossbar Rotate, Crossbar Expand, and Crossbar Shift instructions. As shown in Figs. 32B and 32C, in this exemplary embodiment, the contents of register rc are partitioned into groups of operands of the size specified, and compressed, expanded, rotated or shifted by an amount specified by a portion of the contents of register rb, yielding a group of results. The group of results is catenated and placed in register rd.

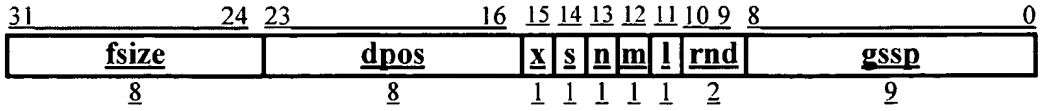
Various Group Compress operations may convert groups of operands from higher precision data to lower precision data. An arbitrary half-sized sub-field of each bit field can be

selected to appear in the result. For example, Fig. 32D shows an X.COMPRESS rd=rc,16,4 operation, which performs a selection of bits 19..4 of each quadlet in a hexlet. Various Group Shift operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in Fig. 32C, certain Group Shift Left instructions may also involve clearing (to zero) empty low order bits associated with the shift, for each operand. Certain Group Shift Right instructions may involve clearing (to zero) empty high order bits associated with the shift, for each operand. Further, certain Group Shift Right instructions may involve filling empty high order bits associated with the shift with copies of the sign bit, for each operand.

Extract

In one embodiment of the invention, data handling operations may also include a Crossbar Extract instruction. Figs. 33A and 33B illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Crossbar Extract instruction. As shown in Figs. 33A and 33B, in this exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

The Crossbar Extract instruction allows bits to be extracted from different operands in various ways. Specifically, bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction (see appendix). The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.:



The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	reserved
m	1	merge vs. extract
l	1	reserved
rnd	2	reserved
gssp	9	group size and source position

The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

The values in the **s**, **n**, **m**, **l**, and **rnd** fields have the following meaning:

values	s	n	m	l	rnd
0	unsigned		extract		
1	signed		merge		
2					
3					

As shown in Fig. 33C, for the X.EXTRACT instruction, when m=0, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra. As shown in Fig. 33D, for a crossbar-merge-extract (X.EXTRACT when m=1), the parameters are interpreted to merge a fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra.

Shuffle

As shown in Fig. 34A, in one embodiment of the invention, data handling operations may also include various Shuffle instructions, which allow the contents of registers to be partitioned into groups of operands and interleaved in a variety of ways. Figs. 34B and 34C illustrate an

exemplary embodiment of a format and operation codes that can be used to perform the various Shuffle instructions. As shown in Figs. 34B and 34C, in this exemplary embodiment, one of two operations is performed, depending on whether the **rc** and **rb** fields are equal. Also, Fig. 34B and the description below illustrate the format of and relationship of the **rd**, **rc**, **rb**, **op**, **v**, **w**, **h**, and **size** fields.

In the present embodiment, if the **rc** and **rb** fields are equal, a 128-bit operand is taken from the contents of register **rc**. Items of size **v** are divided into **w** piles and shuffled together, within groups of **size** bits, according to the value of **op**. The result is placed in register **rd**.

Further, if the **rc** and **rb** fields are not equal, the contents of registers **rc** and **rb** are catenated into a 256-bit operand. Items of size **v** are divided into **w** piles and shuffled together, according to the value of **op**. Depending on the value of **h**, a sub-field of **op**, the low 128 bits (h=0), or the high 128 bits (h=1) of the 256-bit shuffled contents are selected as the result. The result is placed in register **rd**.

As shown in Fig. 34D, an example of a crossbar 4-way shuffle of bytes within hexlet instruction (X.SHUFFLE.128 rd=rcb,8,4) may divide the 128-bit operand into 16 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The 4 partitions are perfectly shuffled, producing a 128-bit result. As shown in Fig. 33E, an example of a crossbar 4-way shuffle of bytes within triclet instruction (X.SHUFFLE.256 rd=rc,rb,8,4,0) may catenate the contents of rc and rb, then divides the 256-bit content into 32 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The low-order halves of the 4 partitions are perfectly shuffled, producing a 128-bit result.

Changing the last immediate value **h** to 1 (X.SHUFFLE.256 rd=rc,rb,8,4,1) may modify the operation to perform the same function on the high-order halves of the 4 partitions. When **rc** and **rb** are equal, the table below shows the value of the **op** field and associated values for **size**, **v**, and **w**.

op	size	v	w
0	4	1	2
1	8	1	2
2	8	2	2
3	8	1	4
4	16	1	2
5	16	2	2
6	16	4	2
7	16	1	4
8	16	2	4
9	16	1	8
10	32	1	2
11	32	2	2
12	32	4	2
13	32	8	2
14	32	1	4
15	32	2	4
16	32	4	4
17	32	1	8
18	32	2	8
19	32	1	16
20	64	1	2
21	64	2	2
22	64	4	2
23	64	8	2
24	64	16	2
25	64	1	4
26	64	2	4
27	64	4	4

op	size	v	w
28	64	8	4
29	64	1	8
30	64	2	8
31	64	4	8
32	64	1	16
33	64	2	16
34	64	1	32
35	128	1	2
36	128	2	2
37	128	4	2
38	128	8	2
39	128	16	2
40	128	32	2
41	128	1	4
42	128	2	4
43	128	4	4
44	128	8	4
45	128	16	4
46	128	1	8
47	128	2	8
48	128	4	8
49	128	8	8
50	128	1	16
51	128	2	16
52	128	4	16
53	128	1	32
54	128	2	32
55	128	1	64

When **rc** and **rb** are not equal, the table below shows the value of the **op**$_{4..0}$ field and associated values for **size**, **v**, and **w**: **Op**$_5$ is the value of **h**, which controls whether the low-order or high-order half of each partition is shuffled into the result.

op$_{4..0}$	size	v	w
0	256	1	2
1	256	2	2
2	256	4	2
3	256	8	2
4	256	16	2
5	256	32	2
6	256	64	2
7	256	1	4
8	256	2	4
9	256	4	4
10	256	8	4
11	256	16	4
12	256	32	4
13	256	1	8
14	256	2	8
15	256	4	8
16	256	8	8
17	256	16	8
18	256	1	16
19	256	2	16
20	256	4	16
21	256	8	16
22	256	1	32
23	256	2	32
24	256	4	32
25	256	1	64
26	256	2	64
27	256	1	128

Conclusion

Having fully described a preferred embodiment of the invention and various alternatives, those skilled in the art will recognize, given the teachings herein, that numerous alternatives and equivalents exist which do not depart from the invention. It is therefore intended that the invention not be limited by the foregoing description, but only by the appended claims.

ABSTRACT OF THE DISCLOSURE

A programmable processor that comprises a general purpose processor architecture, capable of operation independent of another host processor, having a virtual memory addressing unit, an instruction path and a data path; an external interface; a cache operable to retain data communicated between the external interface and the data path; at least one register file configurable to receive and store data from the data path and to communicate the stored data to the data path; and a multi-precision execution unit coupled to the data path. The multi-precision execution unit is configurable to dynamically partition data received from the data path to account for an elemental width of the data and is capable of performing group floating-point operations on multiple operands in partitioned fields of operand registers and returning catenated results. In other embodiments the multi-precision execution unit is additionally configurable to execute group integer and/or group data handling operations